Le langage vectoriel Hellena

Yvon Jégou

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 703

LE LANGAGE VECTORIEL HELLENA

Yvon JEGOU

JUILLET 1987

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex: UNIRISA 950 473 F
Télécopie: 99 38 38 32

Le langage vectoriel Hellena

The vector language Hellena

Publication Interne n° 368 - 60 Pages - Juin 1987

Yvon JEGOU

IRISA / INRIA
Campus Universitaire de Beaulieu
35042 RENNES CEDEX

Résumé

Hellena est un langage de programmation vectoriel, c'est-à-dire un langage dans lequel le domaine d'application des opérateurs scalaires est étendu aux vecteurs et plus généralement aux tableaux de scalaires. Le concept de *patrons d'accès* du langage Hellena permet d'identifier une grande variété de vecteurs et de tableaux dans les structures de données des programmes. Ce concept original élargit le domaine d'application des opérateurs vectoriels et des sous-programmes. Les constructions proposées dans Hellena sont indépendantes de toute architecture d'ordinateur et doivent permettre une programmation efficace d'une large famille de super-calculateurs existants ou en projet.

Abstract

Hellena is a vector language, ie a language which scalar operators can be applied directly to vectors and to arrays. The concept of acces patterns of Hellena allows the identification of numerous vectors and arrays in the data structures of the programs. This original concept increases the application domain of the vector instructions and of the subprograms. Hellena is independant of computer architectures, and should allow an efficient programming of a large familly of existant or projected supercomputers.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

Présentation

Le langage Hellena est un langage du type vectoriel, c'est à dire un langage algorithmique dans lequel tous les opérateurs scalaires classiques peuvent être appliqués directement sur des vecteurs et plus généralement sur des tableaux de scalaires. Par le concept de *patrons d'accès*, le langage Hellena permet d'identifier de nombreux vecteurs et tableaux dans les objets existants des programmes, et donc d'élargir le champs d'application des opérateurs et des sous-programmes. Bien que la cible privilégiée de Hellena soit le calculateur OPSILA (architecture SIMD/SPMD développée au LASSY, Université de NICE), les constructions proposées dans ce langage sont indépendantes de toute architecture d'ordinateur et doivent permettre une programmation efficace d'une large famille de super calculateurs existants ou en projet.

1 Les opérations vectorielles, les tableaux conformants

En Hellena, tous les opérateurs scalaires peuvent être automatiquement appliqués à des vecteurs et des tableaux **conformants**. Deux tableaux sont dits conformants lorsqu'ils ont le même nombre de dimensions et le même nombre d'éléments sur les mêmes dimensions. Ainsi, le domaine d'application d'un opérateur O défini sur des scalaires par

```
O :  s1 x  s2 → s3
```

est automatiquement étendu à des tableaux par :

```
O :  s1 x ts2 → ts3
O : ts1 x  s2 → ts3
O : ts1 x ts2 → ts3
```

où ts1, ts2 et ts3 sont des tableaux 2 à 2 conformants. Le résultat est calculé par application terme à terme de l'opérateur aux éléments des tableaux. Par exemple, V1, V2, V3 étant des vecteurs conformants de n éléments et λ un scalaire, l'instruction

```
V1:= V2 + λ*V3 ;
```

a la signification suivante :

$\forall\ i \in \{0,\ldots,n-1\}$, V1(i1+i) := V2(i2+i) + λ*V3(i3+i) ;

où i1, i2, i3 sont les premiers indices des vecteurs V1, V2 et V3. Pour éviter toute confusion, nous raisonnons en terme de rang de l'élément sur une dimension ($i^{ième}$ élément) et non en valeur absolue de l'indice de l'élément qui dépend des bornes d'indiçage associées aux vecteurs et qui ne sont pas obligatoirement les mêmes pour tous les tableaux.

Tous les opérateurs scalaires prédéfinis de Hellena sont automatiquement étendus de cette

manière et, contrairement à ce qui se passe dans certains langages comme Pascal ou Ada, les opérations de comparaisons produisent des tableaux conformants d'éléments booléens et non des scalaires. Ainsi, l'expression `V1<V2`, `V1` et `V2` ayant n éléments, produit un vecteur de n booléens. Les seuls opérateurs nouveaux de Hellena sont les opérateurs de réduction, c'est à dire ceux qui, appliqués à un vecteur, produisent un résultat scalaire. C'est le cas de l'opérateur `Somme` et l'expression `Somme (V1*V2)` a comme résultat un scalaire (réel ou entier) et calcule le produit scalaire des deux vecteurs `V1` et `V2`.

L'objectif principal de l'introduction des instructions vectorielles (et matricielles) est de permettre la génération des instructions vectorielles des calculateurs SIMD et/ou pipelines. Comme l'efficacité de l'exécution d'un code sur un calculateur vectoriel dépend d'abord de la quantité d'instructions vectorielles exécutées, il est évident que l'efficacité d'un programme Hellena dépend de la quantité d'instructions vectorielles du programme.

2 Structure générale d'un programme

Comme en Ada et en Pascal et au contraire de Fortran, un programme Hellena a une structure de blocs. Un bloc est constitué d'une partie déclarative des entités locales et d'un corps définissant l'algorithme réalisé par l'exécution du bloc. Le programme principal, les procédures, les fonctions et l'instruction bloc sont des blocs. Une partie déclarative est une suite de paragraphes.

- les paragraphes **constante** contiennent des déclarations de constantes ;
- les paragraphes **valeur** contiennent des déclarations de valeurs ;
- les paragraphes **type** contiennent des déclarations de types ;
- les paragraphes **variable** contiennent des déclarations de variables ;
- les paragraphes **patrons** contiennent des déclarations de patrons ;
- les paragraphes **procédure** et **fonction** contiennent respectivement des déclarations de procédures et de fonctions ;
- les paragraphes **opérateur** contiennent des déclarations d'opérateurs ;
- les paragraphes **étiquette** contiennent les déclarations des étiquettes. Elles permettent de nommer les instructions du programme. L'association de l'étiquette à une instruction se fait par une notation de la forme

```
? NOM_étiquette instruction
```

Tous ces paragraphes peuvent apparaître dans n'importe quel ordre et plusieurs fois dans une même partie déclarative. La seule règle imposée par Hellena est la suivante : un identificateur doit avoir été déclaré avant toute utilisation. En règle générale, toute déclaration d'une partie déclarative ou de paramètres a la forme suivante :

```
liste_identificateurs : définition ;
```

Toute déclaration se termine par un ";" même pour la déclaration du dernier paramètre d'une liste

de paramètres formels. Une déclaration devient effective, c'est-à-dire les identificateurs déclarés deviennent accessibles, après le symbole ";" qui signale la fin de la déclaration. Lorsque plusieurs identificateurs sont définis par une même déclaration, cette déclaration produit le même effet qu'une suite de déclarations de même définition. La seule différence réside dans le point où les identificateurs deviennent accessibles. Les entités d'un programme Hellena sont les constantes, les types, les variables, les valeurs, les résultats, les procédures, les fonctions, les patrons d'accès et les opérateurs. Certaines entités peuvent être identifiées par un nom.

3 Les noms

Les noms désignent des entités d'un programme. Le nom d'une constante, d'un type, d'un patron, d'un opérateur, d'une variable, d'une valeur, d'un résultat, d'une procédure ou d'une fonction déclaré dans un programme est l'identificateur associé à sa déclaration dans une partie déclarative ou dans une déclaration des paramètres d'une procédure, d'une fonction, d'un patron ou d'un opérateur. Par exemple, la suite de déclarations {1} définit deux noms : T désigne un type et M désigne une variable.

```
      type
         T : tableau IND de réel ;
      variable
         M : T ;
{1}
```

Les notations d'attributs, de paramétrage, d'application de patrons d'accès, et de sélection permettent de construire de nouveaux noms. Un nom a donc la syntaxe générale définie en {2}.

```
      nom ::= identificateur
            | nom . identificateur_d_attribut
            | nom . identificateur_de_patron
            | nom ` identificateur_de_champ
            | nom ( liste_de_paramètres )
{2}
```

3.1 les notations d'attributs

Un attribut caractérise une entité d'un programme, et dépend de cette entité. Les principaux attributs prédéfinis sont :

a) sur un type T

- si T est un type scalaire discret, les attributs prédéfinis sont `binf`, `bsup`, `cardinal` et

dimensions.

T.binf désigne la plus petite valeur appartenant au type T.
T.bsup désigne la plus grande valeur appartenant au type T.
T.cardinal est une valeur entière et désigne le nombre de valeurs appartenant à T.
T.dimensions désigne la constante entière 1.

T.binf et T.bsup sont du type de base de T.

• si T est un type scalaire non discret (réel), seul l'attribut dimensions est défini et désigne la constante entière 1.

• si T est un type de n-uplets discrets, les attributs gauche, droite, cardinal et dimensions sont définis. Un n-uplet est construit par des couples avec parenthésage à gauche (pour être en conformité avec les règles de précédence habituelles des langages de programmation). Par exemple, le type T : t1*t2*t3 est construit par parenthésage à gauche (t1*t2)*t3 de la manière suivante :

```
t' : t1*t2 ;
 T : t'*t3 ;
```

Les deux attributs gauche et droite d'un type de couples T identifient deux types tels que :

T ≡ T.gauche * T.droite

et les valeurs de la forme (x,y) appartenant à T sont telles que x ∈ T.gauche et y ∈ T.droite.
Un type de triplets T3 est tel que

T3 ≡ T3.gauche.gauche * T3.gauche.droite * T3.droite

L'attribut cardinal d'un type T de n-uplets discrets est une valeur entière telle que

T.cardinal = T.gauche.cardinal * T.droite.cardinal

L'attribut dimensions est défini par

T.dimensions = T.gauche.dimensions + 1

• si T est un type de n-uplets non discrets, les attributs préfinis de T sont gauche, droite et dimensions et ont la même définition que pour les types de n-uplets discrets.

• si T est un type de tableau, l'attribut dimensions désigne la constante entière 1 et les attributs indices et éléments identifient des types tels que

```
T ≡ tableau T.indices de T.éléments
```

b) **sur une valeur, un résultat ou une variable**

Les variables, valeurs et les résultats possèdent un seul attribut : `type_` qui désigne le type de la valeur, de la variable ou du résultat.

3.2 l'application des patrons

Les patrons et leur application sont traités en 10.1.

3.3 la notation de paramétrage

Cette notation de la forme `nom (paramètres)` permet d'indicer les objets tableaux, d'instancier les patrons paramétrés, ou de passer les paramètres dans les appels de procédures et de fonctions.

a) **l'indiçage**

Si le nom désigne un objet tableau, cette notation permet d'identifier un élément du tableau. Pour celà, la notation d'indice est considérée comme un n-uplet qui doit appartenir au type des indices du type associé au tableau. Par exemple, si `M` est déclaré

```
M : tableau [a,b]*[c,d] de réel ;
```

la notation `M(i,j)` identifie l'élément `(i,j)` de `M` à condition que ce couple `(i,j)` appartienne au type `[a,b]*[c,d]`. Notons que ce n'est pas l'élément de la $i^{ième}$ ligne, $j^{ième}$ colonne qui est identifié mais l'élément de la `i-a+1`^{ième} ligne, `j-c+1`^{ième} colonne. La définition des bornes inférieures des indices des tableaux est prise en considération uniquement dans ces notations d'indiçage. Le type de l'élément identifié est le type d'élément défini par le type de l'entité :

```
M(i,j).type_ ≡ M.type_.éléments
```

Si le nom auquel la notation d'indiçage est appliqué est un nom de valeur, de variable ou de résultat, l'élément identifié est respectivement une valeur, une variable ou un résultat.

b) **le passage de paramètres**

Si le nom désigne une application de patron paramétré, une procédure, une fonction, cette notation est un passage de paramètres. Les règles de passage de paramètres sont décrites en 11.1 pour les procédures et fonctions, en 10.3 pour les patrons.

3.4 la notation de sélection

Cette notation permet d'accéder aux composantes des objets et des types *structures* et d'identifier des objets SPMD en dehors de ce mode. Le mode SPMD est traité en 12.

Appliquée à un type de structure T, la notation T`x désigne le type du champ x du type structure T. Par exemple, soient les déclarations {3}, T`x désigne le type entier, T`y désigne le type réel.

```
      type
         T : ( x : entier; y : réel; ) ;

      variable
         V : T ;
{3}
```

Appliquée à un objet S de type structure, la notation S`x désigne le composant x de l'objet S. La notation de sélection V`x ({3}) désigne le composant x de V qui est une variable entière. Comme c'est le cas pour la notation d'indiçage, le composant sélectionné a le même statut (variable, valeur ou résultat) que l'objet *structure*.

Appliquée à un objet de type *tableau de structure*, la notation de sélection désigne un objet de type *tableau* dont la définition des indices est extraite du type *tableau de structure* et dont les éléments sont du type du composant sélectionné. Par exemple, soit la suite de déclarations {4}.

```
      type
         complexe : ( re, im : réel; ) ;
      variable
         TC : tableau [1,100] de complexe ;
{4}
```

- la notation TC(i) désigne une variable de type complexe : c'est un couple de réels.
- la notation TC(i)`im désigne un réel : c'est la partie imaginaire de l'élément i du tableau de complexes TC.
- la notation TC`im désigne un vecteur de 100 réels : c'est le vecteur des parties imaginaires du tableau de complexes TC.

```
TC`im.type_ ≡ tableau TC.type_.indices de TC.type_.elememts`im
```

L'intérêt de la généralisation de la notation de sélection aux tableaux de structures est évidente en programmation vectorielle. Pour plus de précision, se reporter à 4.5.

4 Les types

Nous distinguons les types effectifs des types formels en Hellena. Les types formels sont des types incomplètement définis. Leur intérêt est de permettre la généralisation des définitions de procédures, de fonctions, de patrons et d'opérateurs. Cependant, le type des objets effectivement construits doit être complètement spécifié et donc être effectif. Par exemple, il est interdit d'utiliser une spécification de type formel dans une déclaration de variable.

Certains types effectifs de base comme `entier`, `réel`, `booléen`, `caractère` et certains types formels comme `discret`, `scalaire`, `numérique`, `un_entier` sont prédéfinis en Hellena. De nouveaux types effectifs ou formels peuvent être définis par des constructeurs d'intervalles, d'énumération, de n-uplets, de tableaux et de structures. En Hellena, contrairement à Pascal ou Ada, il faut raisonner en terme d'appartenance de valeurs à des types (voir 1). A chaque objet ou valeur d'un programme est associé un type. C'est, soit le type associé à la définition de l'objet ou de la valeur, soit un type déduit du contexte de définition de l'objet ou de la valeur. Le type de base d'un objet est généralement son type associé auquel certaines contraintes de valeurs ont été supprimées. Tous les types prédéfinis sont des types de base.

4.1 le constructeur d'intervalles

La notation `[a,b]` désigne un ensemble de valeurs comprises entre deux bornes a et b. Pour être valide, il faut que les valeurs de bornes appartiennent à un même type de base scalaire et discret (et ordonné). Le type défini par une notation d'intervalle est inclus dans le type de base de ses bornes. Lorsque la borne supérieure est inférieure strictement à la borne inférieure, l'intervalle est vide (`cardinal=0`). Les bornes ne sont pas nécessairement des constantes. Le type de base de l'intervalle est le type de base de ses bornes.

4.2 le constructeur d'énumération

Le constructeur d'énumération permet de construire un nouveau type de base par énumération de son ensemble de valeurs {5}. Ce type est scalaire discret et ordonné (ordre de

```
type
   semaine : énumération lundi, mardi, mercredi, jeudi, vendredi,
                         samedi, dimanche.
```
{5}

définition). On peut donc en extraire des intervalles {6}. Les opérateurs qui peuvent être appliqués à des valeurs de type énumération sont définis en 6.1.

```
type
   travail : [lundi, vendredi] ;
```
{6}

4.3 le constructeur de n-uplets

La notation t1*t2*...*tn dans laquelle t1, t2, ..., tn sont des types définit un type de n-uplets. Une valeur appartenant à un n-uplet est notée (v1, v2, ..., vn), v1 appartenant à t1, v2 à t2, ..., vn à tn. Lorsque tous les types composant un type de n-uplet sont discrets, le type de n-uplets est discret. Le type de base d'un type de n-uplets est un type de n-uplets de même structure construit sur les types de base des composants. Aucun opérateur du langage n'est actuellement défini sur les n-uplets.

4.4 le constructeur de tableaux

Le constructeur de tableaux permet de définir des ensembles de valeurs structurées régulièrement sous forme de tableaux {7}. Dans cette construction, indices désigne un type

```
tableau indices de éléments
```
{7}

discret et éléments désigne le type des éléments qui peut être quelconque. Par exemples,

- **tableau** [1,n] **de** réel désigne l'ensemble des vecteurs à n éléments réels.

- **tableau** [1,100]*[1,200] **de** entier désigne l'ensemble des tableaux à 100 lignes, 200 colonnes d'éléments de type entier.

- **tableau** [1,100] **de tableau** [1,200] **de** entier désigne l'ensemble des vecteurs de 100 éléments qui sont des vecteurs de 200 éléments entiers. Cet ensemble est disjoint du précédent.

Il faut noter que, bien que la définition des indices indique un ensemble de valeurs, seules leurs dimensions et leur cardinal sont pris en considération dans la définition du type. Le type de base associé à un type tableau est un type tableau construit en prenant la même définition d'indices et le type de base des éléments.

4.5 le constructeur de structure

Dans une définition de type, la notation

```
(id1 : type1; id2 : type2; ...; idn : typen; )
```

définit un type de structure. Les valeurs appartenant à ce type sont des n-uplets du type

```
type1 * type2 * ... * typen
```

Par exemple, soit la déclaration de type {8} et la déclaration de variable {9}, C identifie un

```
type
   complexe : (re, im : réel; ) ;
```
{8}

```
variable
   C : complexe ;
```
{9}

espace mémoire suffisant pour stocker des couples de réels. Par la notation de sélection, les composants C`im et C`re identifient les deux variables qui composent le complexe C. Par rapport aux n-uplets, l'intérêt des constructeurs de structures est de permettre la sélection explicite de composants sur les objets auxquels ce type est associé (voir 3.4).

4.6 la déclaration de type

Les définitions de type de Hellena peuvent apparaître dans les déclarations de variables du programme, les définitions de paramètres formels de procédures, fonctions et patrons, dans la définition des opérateurs, dans la définition des patrons d'accès et dans les définitions et les déclarations de types. Un type peut être déclaré dans un paragraphe **type** d'une partie déclarative par la notation {10}. Dans la suite du programme, l'identificateur associé désigne le type et peut

```
type
   identificateur : définition_de_type ;
```
{10}

apparaître partout où une définition de type est demandée.

4.7 types formels / types effectifs

Seuls les types effectifs peuvent être utilisés dans les déclarations de variables d'un programme. Les types formels sont incomplètement définis et ne peuvent apparaître que là où des types effectifs leur sont substitués par la suite (voir 10.3 pour les patrons, 7 pour les opérateurs et 11.1 pour les procédures et les fonctions).

Les types effectifs prédéfinis de Hellena sont entier, réel, booléen, caractère. Le

constructeur d'énumération produit des types effectifs. Le constructeur d'intervalle produit des types effectifs lorsque les bornes sont effectives. Un type tableau est effectif si et seulement si le type des indices et le type des éléments sont tous deux effectifs ; il est formel sinon. Un type de n-uplets ou de structure est effectif si tous les types composant la définition sont effectifs ; formel sinon.

Les types formels prédéfinis sont :

un_entier : Ce type est inclus dans le type effectif prédéfini `entier` mais ses bornes de valeurs sont inconnues.

numérique : C'est l'union des types effectifs de base `entier` et `réel`.

scalaire : C'est l'union des types non composés, c'est à dire des types prédéfinis `entier`, `réel`, `booléen` , `caractère` et des types construits par énumération.

discret : C'est l'union des types prédéfinis `entier`, `booléen` et `caractère` et des types définis par énumération. Bien qu'ils soient également discrets, les n-uplets de types discrets n'appartiennent pas à ce type formel.

4.8 le type des objets en Hellena

Comme en Pascal et en Ada, un type bien défini est associé à chaque objet d'un programme Hellena. Certaines constructions du langage Hellena entrainent des vérifications d'appartenance des objets des programmes à certains types. Ainsi, la valeur prise par un objet A appartient à un type T si les conditions suivantes sont vérifiées :

- si T est le type prédéfini `entier`, `booléen`, `caractère`, le type associé à A doit être, soit ce type, soit un intervalle de ce type.

- si T est un type d'énumération, le type associé à A doit être, soit ce type d'énumération, soit un intervalle de ce type.

- si T est un intervalle d'entier, de booléen, de caractère ou d'énumération, le type de base associé à A doit être le type de base de T. De plus, la valeur de A doit être comprise entre les bornes spécifiées par T. Cette vérification se fait à l'exécution car le domaine des valeurs de A n'est pas nécessairement inclus dans T.

- si T est un type tableau, les valeurs prises par A doivent être structurées suivant le même nombre de dimensions et posséder le nombre d'éléments spécifié par T, leurs éléments doivent appartenir au type de base des éléments de T. Lorsque le type des éléments de T est un intervalle, le contrôle d'appartenance des éléments de A à cet intervalle est fait à l'exécution.

- si T est un type de n-uplet ou de structure, les valeurs de A doivent être des n-uplets dont les composants appartiennent aux types correspondants dans T.

Les mêmes règles s'appliquent lorsque T est un type formel. Notons que les types `entier` et

réel sont disjoints.

Les vérifications d'appartenance sont faites à la compilation pour les types de base et sont précisées à l'exécution dans le cas des intervalles.

Le type associé à un objet est, soit défini par l'utilisateur dans une déclaration, soit déterminé par le langage, principalement pour les expressions. Une association explicite d'un type à un objet permet l'application de notations d'indiçage pour les tableaux et de sélection pour les structures.

5 Les déclarations de constantes, de valeurs, de types et de variables

Les déclarations de constantes, de valeurs, de types et de variables sont rassemblées dans les parties déclaratives des programmes.

- une déclaration de constante se fait dans un paragraphe **constante** et prend la forme {11}.

```
      identificateur : expression_constante ;
{11}
```

Sa valeur doit être calculée à la compilation, et donc ne dépendre que de constantes. Cette expression peut dépendre de littéraux, de constantes déjà définies et d'attributs constants {12}.

```
      constante
         un : 1 ;
         volume : T.indices.cardinal ;
{12}
```

Les littéraux d'énumération sont des constantes.

- une déclaration de valeur se fait dans un paragraphe **valeur** et prend la forme {13}.

```
      identificateur : expression ;
{13}
```

L'identificateur désigne le résultat du calcul de l'expression à ce point du programme. Les valeurs des opérandes de l'expression peuvent changer dans la suite du programme, mais ne modifient pas la valeur identifiée.

```
      valeur
         pivot : M(i,j) ;
         inv_pivot : 1/pivot ;
{14}
```

• Une déclaration de type se fait dans un paragraphe **type** et a l'une des formes {15}. Dans la

```
          identificateur : nom_de_type ;
          identificateur : définition_de_type ;
{15}
```

```
       type
          T1 : tableau [1,100] de réel ;
          T2 : ( x : entier ; y : réel ; ) ;
          T3 : T1.indices ;
          T4 : V.type_ ;
          T5 : T2`x ;
{16}
```

suite du programme, l'identificateur désigne un type, c'est-à-dire un ensemble de valeurs.

• une déclaration de variable se fait dans un paragraphe **variable** d'une partie déclarative et prend l'une des trois formes {17}, {18} ou {19}. Les formes (1) et (2) provoquent la réservation en

```
{17}      identificateur : nom_de_type ;
{18}      identificateur : définition_de_type ;
{19}      identificateur : nom_de_variable ;
```

mémoire de l'espace nécessaire à la mémorisation d'une valeur appartenant au type. Pour que cette réservation soit possible, le type associé doit être effectif (voir 4.7).
La forme {19} permet d'identifier une variable existante, et ne provoque pas de réservation d'espace. Par exemple, soit la suite de déclarations {20}, la valeur i n'étant pas nécessairement connue à la compilation. Cette suite de déclarations a les effets suivants :

```
       type
          complexe : (re, im : réel ; ) ;
          tab_complexe : tableau [1,100] de complexe ;

       variable
          TC :    tab_complexe ;
          eTCi :  TC(i) ;
          imTCi : TC(i)`im ;
          imTC :  TC`im ;
          x :     TC`im(i) ;
          y :     eTCi`im ;
{20}
```

- la déclaration de TC provoque l'allocation de l'espace mémoire nécessaire pour stocker un tableau de 100 complexes. TC identifie ce tableau.

- eTCi identifie l'élément i du tableau TC. C'est une variable complexe.

```
eTCi.type_ ≡ complexe
```

  i est une expression quelconque dont la valeur appartient au domaine d'indiçage de TC. C'est la valeur de i au moment de l'identification qui est utilisée. La valeur de i peut changer dans la suite du programme, mais ne provoque pas de changement d'élément identifié.

- imTCi identifie le composant im de l'élément i du tableau de complexes TC. C'est donc une variable réelle.

```
imTCi.type_ ≡ TC.type_.éléments`im ≡ réel
```

- imTC identifie le vecteur des parties imaginaires du tableau de complexes TC. C'est donc une variable de type tableau de réel.

```
imTC.type_ ≡ tableau TC.type_.indices de TC.type_.éléments`im
           ≡ tableau [1,100] de réel
```

- x identifie l'élément i du vecteur des partie imaginaires du tableau de complexes TC. C'est une variable de type réel.

```
x.type_ ≡ TC`im.type_.éléments ≡ réel
```

- y identifie la partie imaginaire de la variable complexe eTCi. C'est une variable réelle.

```
y.type_ ≡ eTCi`im.type_ ≡ réel
```

En résumé, imTCi, x et y désignent la même variable réelle, c'est-à-dire, ces trois variables se partagent le même emplacement mémoire.

6 Les opérateurs prédéfinis

Le langage Hellena prédéfinit deux types d'opérateurs :

- les opérateurs scalaires dont l'application est étendue aux tableaux de scalaires. Certains opérateurs scalaires sont appliqués par une notation de fonction.
- les opérateurs de réduction liés à la nature vectorielle du langage. Tous ces opérateurs sont appliqués par une notation de fonction.

6.1 les opérateurs scalaires

Tous les opérateurs scalaires ({21} et {22}) peuvent être appliqués à des tableaux de

+, -, *					
	entier	x	entier	↦	entier
	entier	x	réel	↦	réel
	réel	x	entier	↦	réel
	réel	x	réel	↦	réel
/					
	entier	x	entier	↦	réel
	entier	x	réel	↦	réel
	réel	x	entier	↦	réel
	réel	x	réel	↦	réel
div, modulo, reste					
	entier	x	entier	↦	entier
=, <>, <, <=, >, >=					
	entier	x	entier	↦	booléen
	entier	x	réel	↦	booléen
	réel	x	entier	↦	booléen
	réel	x	réel	↦	booléen
	booléen	x	booléen	↦	booléen
	caractère	x	caractère	↦	booléen
	énumération	x	énumération	↦	booléen
et, ou, ou_exclusif					
	booléen	x	booléen	↦	booléen
min, max					
	entier	x	entier	↦	entier
	entier	x	réel	↦	réel
	réel	x	entier	↦	réel
	réel	x	réel	↦	réel
	booléen	x	booléen	↦	booléen
	caractère	x	caractère	↦	caractère
	énumération	x	énumération	↦	énumération

{21}

scalaires. Le résultat est alors un tableau conformant (même nombre de dimensions, même nombre d'éléments sur chaque dimension que les opérandes) obtenu par application terme à terme de l'opérateur aux éléments des tableaux. Les bornes de définition des indices des opérandes ne sont pas pris en considération. Les opérations entre scalaires et tableaux de scalaires sont également possibles.

```
-
                        entier   ↦   entier
                         réel    ↦    réel
non
                       booléen   ↦   booléen
absolu
                        entier   ↦   entier
                         réel    ↦    réel
ceil, floor
                         réel    ↦    réel
en_réel
                        entier   ↦    réel
en_entier
                         réel    ↦    entier
successeur, prédécesseur
                        entier   ↦   entier
                       booléen   ↦   booléen
                      caractère  ↦  caractère
                     énumération ↦ énumération
```

{22}

6.2 les opérateurs de réduction

Les opérateurs de réduction {23} s'appliquent à des vecteurs (tableaux à une seule dimension) de scalaires et produisent un résultat scalaire. Ils s'appliquent tous par une notation de fonction.

```
Somme, Produit
                    vecteur d'entiers      ↦ entier
                    vecteur de réels       ↦ réel
Min, Max
                    vecteur d'entiers      ↦ entier
                    vecteur de réels       ↦ réel
                    vecteur de booléens    ↦ boòléen
                    vecteur de caractères  ↦ caractère
                    vecteur d'énumérations ↦ énumération
Union, Intersection
                    vecteur de booléens    ↦ booléen
```

{23}

7 Les définitions d'opérateurs

Il est possible de définir de nouveaux opérateurs en Hellena. Un nouvel opérateur est déclaré dans un paragraphe **opérateur** d'une partie déclarative par la notation {24}. Nous avons

```
      opérateur
      identificateur ( opérandes ) dans type_résultat : expression ;
{24}
```

```
   opérande_gauche "identificateur_opérateur" opérande_droit
{25}
```

limité ces déclarations aux opérateurs binaires. Un opérateur défini est appliqué par une notation d'opérateur de la forme {25}. Par exemple, soient l'opérateur `produit_scalaire` déclaré en {26}, une variable réelle a, deux vecteurs v1 et v2, l'instruction {27} provoque l'application de

```
      opérateur
         produit_scalaire ( valeur x, y : vecteur ; ) dans réel :
            Somme (x*y) ;
{26}
```

```
      a := v1 "produit_scalaire" v2 ;
{27}
```

l'opérateur `produit_scalaire` à v1 et v2 ; le résultat est affecté à la variable a. Une application d'opérateur défini a la priorité la plus faible par rapport aux opérateurs prédéfinis. Les types des paramètres et du résultat d'une définition d'opérateur peuvent être formels. Des règles similaires à celles utilisées dans les patrons sont appliquées pour les substitutions des paramètres et pour déterminer le type du résultat. On peut, par exemple, définir l'opérateur `produit_scalaire` en {28}. Par substition des paramètres effectifs aux paramètres formels, cet

```
   opérateur
      produit_scalaire ( valeur x, y : tableau discret de numérique ; )
                       dans numérique :
         Somme (x*y) ;
{28}
```

opérateur peut être appliqué à des vecteurs de réels, des vecteurs d'entiers, un vecteur de réel à gauche et un vecteur d'entier à droite, un vecteur d'entier à gauche et un vecteur de réel à droite. Le type du résultat est déterminé par l'application des opérateurs prédéfinis de l'expression interne. Notons l'importance des types formels dans cette généralisation.

Dans l'exemple {28}, la définition de l'opérateur n'impose pas que les deux vecteurs opérandes

soient conformes. Les erreurs éventuelles sont généralement détectées dans le corps de l'opérateur. Il est possible d'imposer certaines relations entre les types des paramètres. En effet, comme nous le verrons pour les procédures et les fonctions, dès qu'un paramètre formel est défini, il peut être utilisé dans les définitions des paramètres suivants. Par exemple, on peut imposer que les opérandes de l'opérateur `produit_scalaire` soient conformes {29}. Dans une

```
opérateur
   produit_scalaire ( valeur x : tableau discret de numérique ;
                      y : tableau x.type_.indices de numérique ; )
                     dans numérique :
      Somme (x*y) ;
```
{29}

application de cet opérateur, les règles de substitution des paramètres entraînent une vérification sur le nombre d'éléments de l'opérande droit. Contrairement aux opérateurs prédéfinis, l'application des opérateurs définis sur des scalaires n'est pas automatiquement étendue aux tableaux de scalaires.

Cette notion d'opérateurs n'est pas fondamentale en Hellena. Elle a été introduite pour faciliter certaines écritures de programmes.

8 Les expressions conditionnelles

Une expression conditionnelle permet de produire une valeur en fonction de conditions. Sa forme générale est décrite en {30}. Les conditions sont des expressions à résultat booléen

```
si condition_1
   alors expression_1
sinon_si condition_2
   alors expression_2
...
sinon expression_f
fin_si
```
{30}

scalaire. Toutes les expressions composant une expression conditionnelle doivent produire des résultat appartenant à un même type de base ; ce type de base étant le type du résultat de l'expression conditionnelle. La partie **sinon** est obligatoire. Pour évaluer une expression conditionnelle, les conditions sont calculées dans l'ordre jusqu'à la première vérifiée. Le résultat de l'expression conditionnelle est celui de l'évaluation de l'expression correspondante. Lorsqu'aucune condition n'est vérifiée, le résultat est celui de la partie **sinon**. Par exemple, l'expression {31} produit la valeur absolue de a. L'expression {32} permet d'accéder en lecture à l'élément (i,j) d'une matrice symétrique M dont seul le triangle supérieur est à jour.

```
      si a>0 alors a sinon -a fin_si
```
{31}

```
      si i>j alors M(j,i) sinon M(i,j) fin_si
```
{32}

9 Les expressions pour_tout

Une expression **pour_tout** a la forme générale {33}. L'évaluation d'une expression **pour_tout**

```
      pour_tout i1 dans IND1,
      pour_tout i2 dans IND2,
      ...
      pour_tout in dans INDn :
            expression
      fin_pour_tout
```
{33}

identifie une valeur ou une variable de type tableau de dimensions déterminées par le domaine de définition du **pour_tout** (IND1*IND2*...*INDn) et d'éléments du type de l'expression. Cette construction est intéressante lorsqu'elle est associée à des expressions conditionnelles ; voir les exemples {34}, {35}, {36} et {37}.

```
      pour_tout i dans [1,n] :
         si a(i)<b(i) alors a(i) sinon b(i) fin_si
      fin_pour_tout

                  l'opérateur min appliqué à deux vecteurs a et b
```
{34}

```
      pour_tout j dans [1,n] :
         si i<j alors M(i,j) sinon M(j,i) fin_si
      fin_pour_tout

      La ligne i d'une matrice symétrique M en ne faisant accès qu'au triangle supérieur
```
{35}

```
Somme (pour_tout i dans [2,n] :
         si x(i-1)*x(i)<0 alors 1 sinon 0 fin_si
       fin_pour_tout )
```

Le nombre de changements de signe dans le vecteur x

{36}

```
Somme (pour_tout i dans [1,n] :
         si x(i)>0 alors x(i) sinon 0 fin_si
       fin_pour_tout )
```

La somme des éléments positifs d'un vecteur x

{37}

10 Les patrons

Un patron, défini comme application d'un type de départ dans un type d'arrivée permet d'identifier un nouvel objet du type d'arrivée dont les éléments sont éléments d'un objet du type de départ. Par exemple, soit le tableau M déclaré en {38}, la notation M.diagonale définit une

```
variable
    M: tableau [1,100]*[1,100] de réel ;
```

{38}

variable de type "tableau [1,100] de réel" dont l'élément d'indice i est l'élément d'indice (i,i) de la matrice M : ∀ i, M.diagonale(i) ≡ M(i,i) (figure 1). Un patron est défini par des expressions d'indiçage qui mettent en relation l'élément ou les éléments de l'objet résultat et des éléments de l'objet auquel le patron est appliqué. Le résultat d'une application de patron est un objet du langage Hellena ; il peut être indicé, peut être paramètre effectif de procédure ou de fonction, un patron peut lui être appliqué. Les figures 1 à 8 montrent des exemples de patrons courants dans les problèmes numériques.

10.1 la définition des patrons

Les patrons d'accès les plus fréquemment utilisés pourraient être prédéfinis. Pour permettre l'identification de nouveaux vecteurs et de nouveaux tableaux dans les programmes, et donc d'élargir le champ d'application des instructions vectorielles, le langage Hellena permet la définition de nouveaux patrons. Un patron est déclaré dans un paragraphe **patron** d'une partie déclarative par la notation de la forme {39}. Le type d'entrée et le type du résultat peuvent être formels. Un patron ne peut s'appliquer qu'à des objets dont le type est inclus dans le type d'entrée spécifié.

Définir un patron d'accès consiste à définir les relations entre des valeurs d'indices de

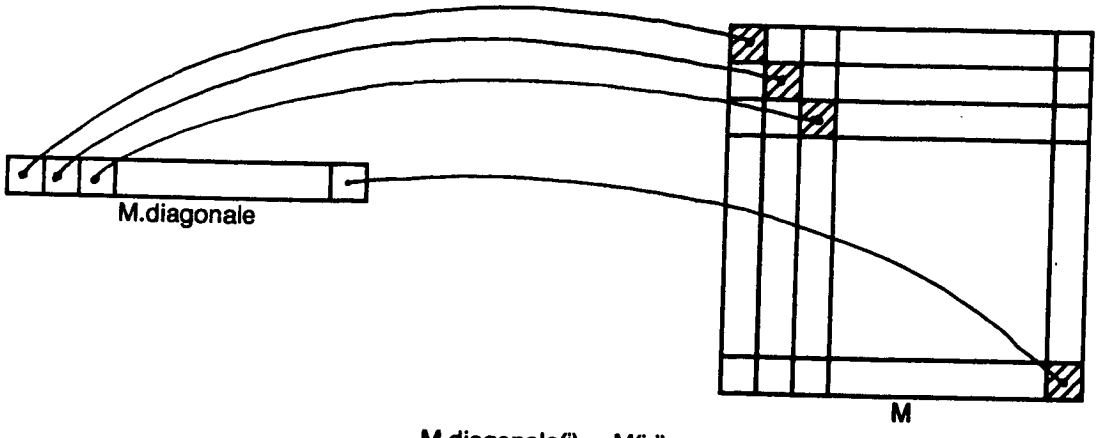


figure 1

```
      patron
         identificateur {(paramètres éventuels)}
              de type_d'entrée dans type_du_résultat :
                   définition_du_patron ;
```
{39}

l'objet résultat et des valeurs d'indices de l'objet auquel le patron est appliqué. Ces relations sont exprimées en Hellena par des expressions d'indiçage de l'objet d'entrée dépendant de générateurs d'indices de l'objet résultat.

- Pour un type résultat de la forme **tableau** D **de** E, le générateur d'indices a la forme {40}.

```
      pour_tout I dans D : F(I) fin_pour_tout
```
{40}

F(I) est la fonction d'indiçage de l'objet en entrée. Cette fonction fait généralement référence aux générateurs d'indices et peut faire référence aux paramètres du patron ainsi qu'à des valeurs externes. Lorsque le domaine d'indices D est un type de n-uplets d1*d2*..*dn, le générateur d'indices doit être développé de la forme {41} pour permettre une définition de la fonction d'indiçage par des expressions sur des valeurs scalaires.

- Pour un type résultat de la forme **tableau** D1 **de tableau** D2 **de** E, le générateur d'indices a la forme {42}. Comme dans la forme précédente, chaque niveau de **pour_tout** peut être développé pour définir la fonction F sur des valeurs scalaires.

```
      pour_tout i1 dans d1,
      pour_tout i2 dans d2,
      ...
      pour_tout in dans dn : f(i1, i2, ..., in) fin_pour_tout
{41}
```

```
      pour_tout I1 dans D1 :
         pour_tout I2 dans D2 : F(I1, I2) fin_pour_tout
      fin_pour_tout
{42}
```

En utilisant ces notations, les patrons d'accès `diagonale` ({43}, figure 1) et `ligne` ({44}, figure 2) peuvent être déclarés sur des tableaux de 100 lignes, 100 colonnes de réels.

```
      patron diagonale de tableau [1,100]*[1,100] de réel
                     dans tableau [1,100] de réel :
         pour_tout i dans [1,100] : (i,i) fin_pour_tout ;
{43}
```

```
      patron ligne de tableau [1,100]*[1,100] de réel
                           dans tableau [1,100] de tableau [1,100] de réel :
         pour_tout i dans [1,100] :
            pour_tout j dans [1,100] : (i,j) fin_pour_tout
         fin_pour_tout  ;
{44}
```

Lorsque le patron `ligne` est appliqué à la matrice `M` déclarée en {45},

- La notation `M.ligne` définit une variable structurée sous forme de vecteur de 100 éléments, ces éléments étant eux-même des vecteurs de 100 réels.
- La notation `M.ligne(i)` permet l'accès au vecteur ligne `i` de `M`.
- La notation `M.ligne(i)(j)` désigne l'élément `M(i,j)` de `M`.

Les applications de patrons peuvent être composés. Par exemple, les notations `M.ligne` et `M.transpose.colonne` identifient un même objet {46}.
Le langage Hellena permet le paramétrage des définitions de patrons {47}. Bien que sémantiquement différente, cette nouvelle définition du patron `ligne` est équivalente à la définition {44}. La différence réside dans le fait que l'objet `M.ligne` de la première définition existe (c'est un tableau de tableau) et la notation d'indiçage n'est pas obligatoire alors que dans la notation `M.ligne` issue de la deuxième définition, le patron `ligne` doit être paramétré. L'intérêt de cette possibilité de paramétrage réside dans le fait que les paramètres peuvent être utilisés dans les définitions des types d'entrée et du résultat, dans les générateurs d'indices ainsi que dans les expressions d'indiçage. Le patron `ligne_sup` défini en {48} permet d'identifier la portion de

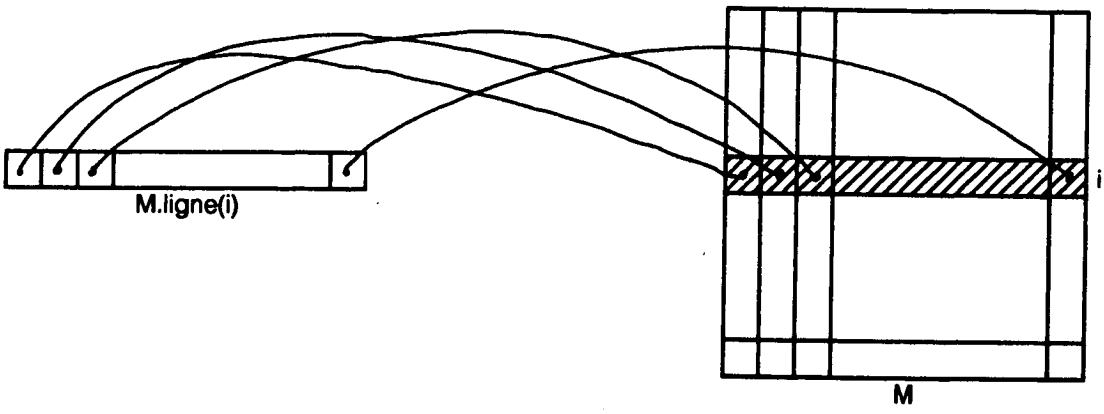


figure 2

```
      variable
         M : tableau [1,100]*[1,100] de réel ;
{45}
```

```
     ∀ i, ∀ j, M.ligne(i)(j) ≡ M.transpose.colonne(i)(j).
{46}
```

ligne correspondant au triangle supérieur d'une matrice (figure 5).

10.2 l'application des patrons

L'application d'un patron provoque le remplacement de la définition de l'opérande formel par l'opérande effectif. Pour celà, il faut que le type de l'opérande effectif soit conforme au type associé au patron. Dans une définition de patron, l'identificateur `type_opérande` désigne le type effectif de l'objet auquel le patron d'accès est appliqué. Si le type associé au patron est formel, `type_opérande` désigne le type de l'opérande effectif. Si le type d'entrée du patron est effectif, `type_opérande` désigne ce type. Ce type, et principalement ses attributs peuvent être utilisés partout dans une déclaration de patron : pour définir le type résultat, dans les générateurs d'indices et dans les expressions d'indiçage.

L'application du corps du patron crée un premier type tableau dont les spécifications d'indices sont déduites des valeurs de contrôle des expressions **pour_tout** et dont les éléments sont du type des éléments de l'opérande. Ce type construit doit être conforme au type résultat

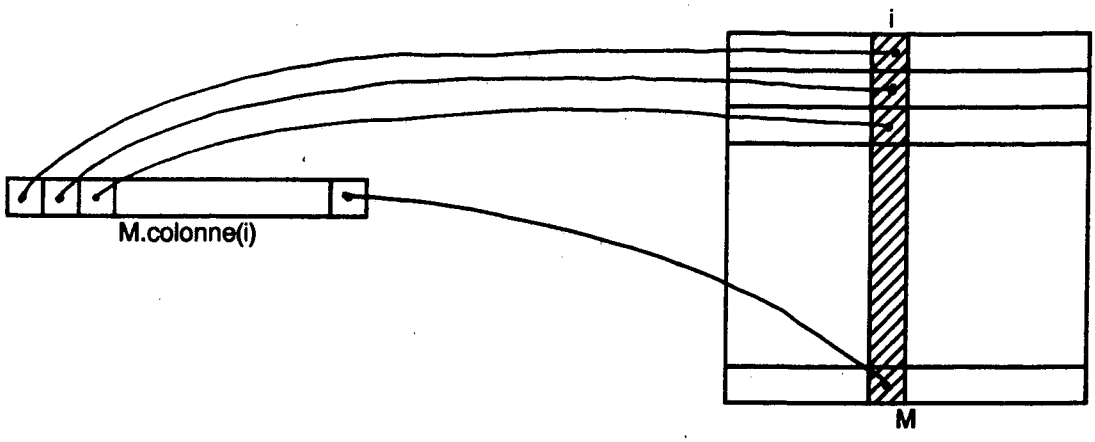


figure 3

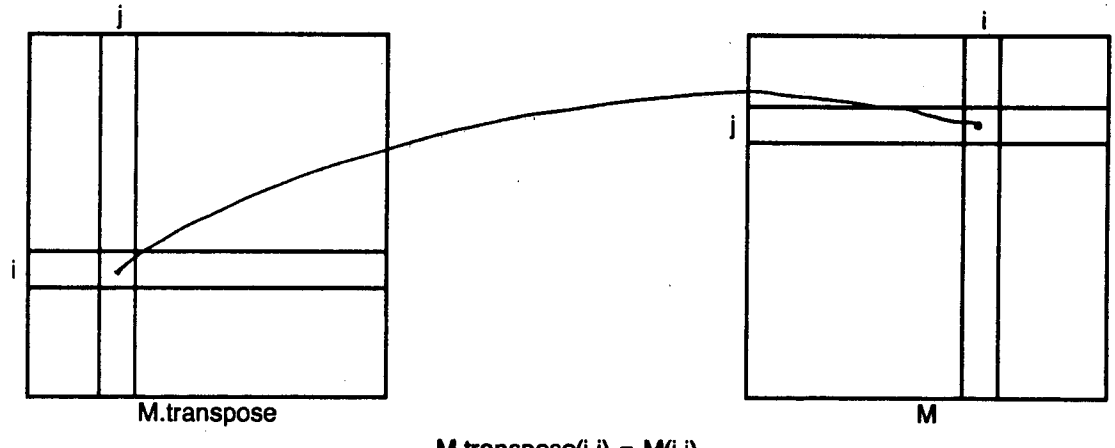


figure 4

```
patron ligne (valeur i : [1,100];)
        de tableau [1,100]*[1,100] de réel
      dans tableau [1,100] de réel :
    pour_tout j dans [1,100] : (i,j) fin_pour_tout ;
```
{47}

déclaré du patron. Le type de l'objet résultat de l'application du patron est

```
patron ligne_sup (valeur i : [1,100];)
        de tableau [1,100]*[1,100] de réel
      dans tableau [1,101-i] de réel :
   pour_tout j dans [i,100] : (i,j) fin_pour_tout ;
```
{48}

o soit le type résultat si c'est un type effectif,

o soit un type conforme construit en remplaçant dans le type résultat les types élémentaires formels par les types correspondants déterminés par l'application du corps.

Ces diverses possibilitées de définition de patrons sur des types formels et d'utilisation des règles de conformités permettent de généraliser de nombreuses définitions. Par exemple, soit la déclaration {49}, ce patron peut être appliqué à tout tableau à deux dimensions d'éléments

```
patron ligne de tableau discret*discret de scalaire
                dans tableau discret de tableau discret de scalaire :
   pour_tout i dans type_opérande.indices.gauche :
      pour_tout j dans type_opérande.indices.droite : (i,j)
      fin_pour_tout
   fin_pour_tout
```
{49}

scalaires. De plus, il conserve la numérotation des lignes et des colonnes du tableau de départ.

Les définitions en {50} traitent les divers cas qui peuvent apparaître quand les règles de substitution sont appliquées. Applications :

M1.diagonale1
type de l'opérande considéré: **tableau** [1,100]*[1,100] **de** réel
type construit : **tableau** [1,100] **de** réel
type résultat : TV (est formel)
type de M1.diagonale1 : **tableau** [1,100] **de** réel

M2.diagonale1
type de l'opérande considéré: **tableau** [0, 99]*[0, 99] **de** réel
type construit : **tableau** [0, 99] **de** réel
type résultat : TV (est formel)
type de M2.diagonale1 : **tableau** [0, 99] **de** réel

```
type
   TM : tableau discret*discret de réel ;
   TM1: tableau [1,100]*[1,100] de réel ;
   TM2: tableau [0, 99]*[0, 99] de réel ;
   TV : tableau discret de réel ;
   TV1: tableau [1,100] de réel ;
   TV2: tableau [0, 99] de réel ;

patron
   diagonale1 de TM dans TV :
      pour_tout i dans type_opérande.indices.gauche : (i,i)
      fin_pour_tout ;

   diagonale2 de TM dans TV :
      pour_tout i dans [1, type_opérande.indices.gauche.cardinal] :
         (i + type_opérande.indices.gauche.binf-1,
          i + type_opérande.indices.gauche.binf-1)
      fin_pour_tout ;

   diagonale3 de TM1 dans TV1 :
      pour_tout i dans [1,100]: (i,i) fin_pour_tout ;

variable
   M1 : TM1 ;
   M2 : TM2 ;
{50}
```

M1.diagonale2
type de l'opérande considéré: **tableau [1,100]*[1,100] de réel**
type construit : **tableau [1,100] de réel**
type résultat : TV (est formel)
type de M1.diagonale2 : **tableau [1,100] de réel**

M2.diagonale2
type de l'opérande considéré: **tableau [0, 99]*[0, 99] de réel**
type construit : **tableau [1,100] de réel**
type résultat : TV (est formel)
type de M2.diagonale2 : **tableau [1,100] de réel**

```
M1.diagonale3
      type de l'opérande considéré: tableau [1,100]*[1,100] de réel
      type construit : tableau [1,100] de réel
      type résultat : tableau [1,100] de réel
      type de M1.diagonale3 : tableau [1,100] de réel

M2.diagonale3
      type de l'opérande considéré: tableau [1,100]*[1,100] de réel
      type construit : tableau [1,100] de réel
      type résultat : tableau [1,100] de réel
      type de M2.diagonale3 : tableau [1,100] de réel
```

10.3 l'application des patrons paramétrés

L'application d'un patron paramétré suit les mêmes règles que l'application d'un patron non paramétré : le patron est d'abord appliqué au tableau sans tenir compte des paramètres formels. Le résultat de cette application est un objet formel qui doit être instancié par passage de paramètres effectifs. Cet ordre d'évaluation est important car il permet de faire dépendre la définition des paramètres formels d'attributs du type de l'opérande. Par exemple, soit la définition de patron {51}, figure 5, ce patron `lignesup` a les caractéristiques suivantes :

```
   patron
      lignesup (valeur i : type_opérande.indices.gauche ; )
           de tableau discret*discret de scalaire
   dans tableau [1, type_opérande.indices.droite.cardinal-i+1] de scalaire :
         pour_tout j dans [i, type_opérande.indices.droite.bsup] :
            (i,j)
         fin_pour_tout ;
{51}
```

- il peut être appliqué à tout tableau à deux dimensions d'éléments scalaires ; les définitions de bornes des indices sont quelconques.

- le résultat est un vecteur (tableau à une dimension) d'indices entiers de borne inférieure 1, et d'éléments scalaires.

- lorsqu'il est appliqué à une matrice, ce patron identifie le vecteur correspondant à une ligne du triangle supérieur de cette matrice (voir figure 5). Noter que cette définition n'impose pas que la matrice soit carrée. Le résultat peut être un vecteur de longueur nulle.

- la notation `M.lignesup` identifie un objet formel : c'est une matrice triangulaire supérieure dont la représentation n'est pas prévue dans Hellena. La seule opération possible sur cet objet est l'instanciation par passage des paramètres effectifs qui correspond à l'identification d'une ligne.

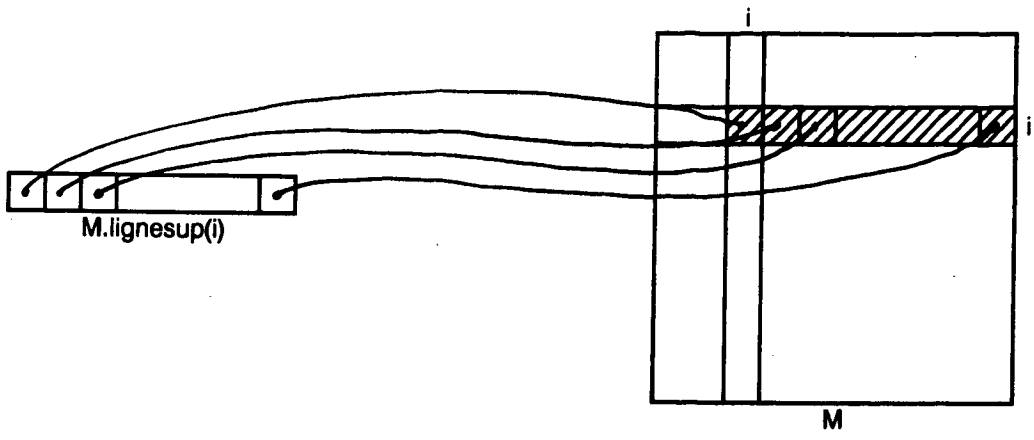


figure 5

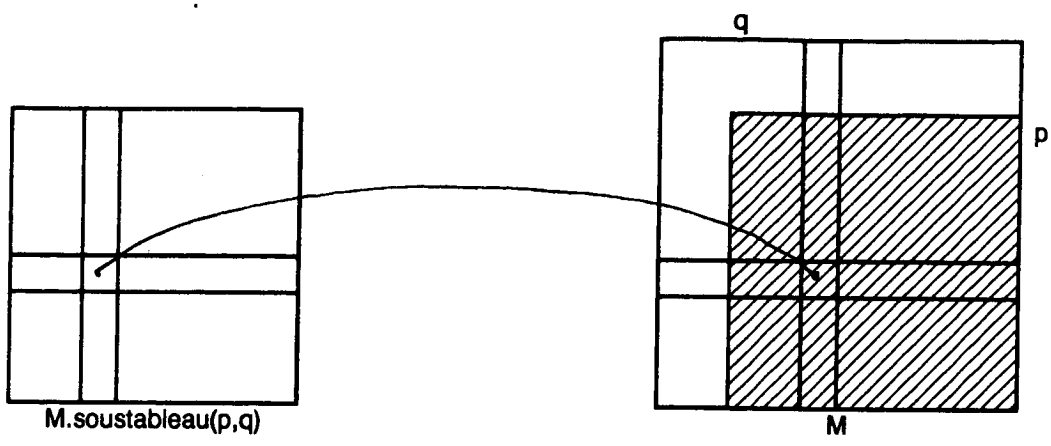
Le type du paramètre formel de cet objet formel est déduit du type des indices de ligne de la matrice à laquelle le patron est appliqué.

L'utilisation de patrons paramétrés est intéressante dans les algorithmes itératifs qui sont appliqués à des sousmatrices de volume dépendant de l'itération (factorisation LU, résolution de systèmes triangulaires, tridiagonalisation d'une matrice symétrique, ...). Elle permet de continuer à programmer en terme d'instructions vectorielles sur des vecteurs (ou des tableaux) de taille variable. Par exemple, le patron `soustableau` de la figure 6 permet d'appliquer l'algorithme F de la boucle {52} sur des soustableaux de taille décroissante.

11 Les procédures et les fonctions

Les procédures et les fonctions de Hellena ont une structure reconnue dans de nombreux langages à structure de bloc. Une procédure est déclarée dans un paragraphe **`procédure`** d'une partie déclarative. Cette déclaration a la forme {53}. La partie déclarative rassemble les déclarations des entités locales de la procédure. C'est une suite de paragraphes **`constante`**, **`type`**, **`variable`**, **`valeur`**, **`opérateur`**, **`patron`**, **`procédure`** et **`fonction`**. Il n'y a pas d'ordre imposé à la succession de ces paragraphes. Un même paragraphe peut apparaître plusieurs fois. Les entités déclarées dans la partie déclarative d'une procédure ne sont accessibles que de la procédure : elles sont locales.

La liste d'instructions définit l'effet de l'exécution de la procédure. Une procédure peut ne pas avoir de paramètres. La liste de déclaration des paramètres est une suite de paragraphes **`constante`**, **`valeur`**, **`variable`**, **`résultat`** et **`type`**. Dès qu'un paramètre formel est déclaré, il devient accessible, même dans la définition des paramètres formels suivants (cette règle a déjà été



M.soustableau(p,q)(i,j) ≡ M(i + p − 1,j + q − 1)

figure 6

```
        pour i := 1 jusqu_a n
        boucle
           F (M.soustableau(i)) ;
        fin_boucle ;
```
{52}

```
        procédure
           identificateur ( paramètres optionnels ) :
              partie déclarative
           début
              liste instructions
           fin
```
{53}

utilisée dans la définition des opérandes d'un opérateur). Tout paramètre formel est déclaré par une notation de la forme {54}.

```
        identificateur : type ;
```
{54}

- les paramètres constante

Les paramètres constante sont déclarés dans les paragraphes **constante** d'une déclaration de

paramètres. A l'appel de la procédure, le paramètre effectif correspondant doit être une constante appartenant au type associé à la déclaration. Dans la procédure, l'identificateur désigne une valeur et non une constante.

- les paramètres valeur

Les paramètres valeur sont déclarés dans les paragraphes **valeur**. A l'appel de la procédure, le paramètre effectif correspondant doit être soit une valeur appartenant au type spécifié, soit une variable dont la valeur courante appartient au type spécifié. Dans le corps de la procédure, l'identificateur désigne une valeur : elle ne peut pas être modifiée par la procédure.

- les paramètres variable

Les paramètres variable sont déclarés dans les paragraphes **variable** des déclarations de paramètres. A l'appel de la procédure, le paramètre effectif associé doit être une variable dont le type inclut le type associé et dont la valeur courante appartient au type associé à la définition du paramètre formel. Dans le corps de la procédure, l'identificateur désigne une variable qui peut être lue et modifiée.

- les paramètres résultat

Les paramètres résultat sont déclarés dans les paragraphes **résultat**. A l'appel de la procédure, le paramètre effectif associé doit être une variable ou un résultat dont le type inclut le type associé à la déclaration. Dans le corps de la procédure, l'identificateur désigne un résultat qui ne peut être qu'écrit.

- les paramètres type

Les paramètres type sont déclarés dans les paragraphes **type**. Le type associé doit être formel. A l'appel de la procédure le paramètre effectif correspondant doit être un type effectif inclus dans le type associé. Dans le corps de la procédure, l'identificateur désigne un type effectif.

11.1 les paramètres de type formel

Le type associé aux paramètres type doivent être formels. Le type associé aux autres paramètres peuvent être effectifs ou formels. Lorsque ce type est effectif, il est associé à la constante, la valeur, la variable ou le résultat déclaré. Cette règle permet de ne pas lier les fonctions d'indiçage de la procédure aux bornes effectives des paramètres effectifs tableau. Tous les types formels ne peuvent pas être utilisés dans les spécifications de paramètres (il faut pouvoir compiler la procédure en dehors de ses contextes d'appels). Seules les spécifications de bornes des intervalles peuvent être indéterminées. Dans le corps de la procédure, tous les types formels de paramètres sont remplacés par des types effectifs dans lesquels les attributs `binf`, `bsup` et `cardinal` identifient les attributs correspondants des paramètres effectifs. Par exemple, soit la déclaration {55} Dans le corps de la procédure P, T désigne une variable effective. `T.type_`

```
      procédure P ( variable T : tableau un_entier de réel ; ) ;
{55}
```

désigne le type du paramètre effectif. Ce type peut être utilisé partout où un type est demandé dans le corps de la procédure. Soit la déclaration {56}, cette procédure peut être la spécification

```
   procédure mxm
    ( valeur A : tableau un_entier*un_entier de réel ;
             B : tableau A.type_.indices.droite*un_entier de réel ;
    variable C : tableau A.type_.indices.gauche*B.type_.indices.droite
                         de réel ;
      ) ;
{56}
```

d'une procédure de multiplication de matrices. A l'appel de cette procédure par la notation `mxm(x,y,z)`, les vérifications suivantes sont faites :

- o x, y et z sont des tableaux à deux dimensions de réels.
- o le nombre de lignes de y est égal au nombre de colonnes de x.
- o le nombre de lignes de z est égal au nombre de lignes de x ; le nombre de colonnes de z est égal au nombre de colonnes de y.

Dans le corps de procédure, les domaines de variation des indices des trois tableaux sont conformes :

- o les indices de lignes de A et de C ont les mêmes bornes,
- o les indices de colonnes de B et de C ont les mêmes bornes,
- o les indices de colonnes de A et les indices de lignes de B ont les mêmes bornes.

Par contre, aucune contrainte n'est imposée aux bornes des paramètres effectifs, ce qui facilite l'appel de cette procédure sur des soustableaux.

11.2 les procédures et les fonctions récursives

Comme pour toute déclaration, l'identificateur d'une procédure ou d'une fonction n'est introduit qu'à la fin de la déclaration. Cette règle interdit toute récursivité des appels de procédures et de fonctions. Une procédure ou une fonction peut cependant être spécifiée récursive comme dans les exemples {57} et {58}.

12 Les déclarations SPMD

Dans une partie déclarative, toutes les déclarations comprises entre les mots clés **spmd** et **fin_spmd** sont des déclarations d'entités SPMD. Une zone de déclarations SPMD est une partie déclarative et peut être composée des paragraphes **constante**, **type**, **valeur**, **variable**,

```
   fonction
      factorielle (valeur n : entier) résultat entier récursive :
      début
         factorielle := si n>1 alors factorielle(n-1)*n sinon 1 fin_si ;
      fin ;
{57}
```

```
   procédure
      tri ( variable X : tableau un_entier de réel ; ) récursive :
      valeur
              milieu : X.type_.indices.binf+X.type_.indices.cardinal div 2 ;
      début
         si X.type_.indices.cardinal>1 alors
            tri(X.portion(X.type_.indices.binf, milieu-1)) ;
            tri(X.portion(milieu, X.type_.indices.bsup)) ;
            fusion(X) ;
         fin_si ;
      fin ;
{58}
```

patron, **procédure** et **fonction** comme toute partie déclarative mais ne peut pas contenir de nouvelle zone SPMD. Dans la version actuelle de Hellena, le SPMD est lié à une architecture particulière : celle d'un multiprocesseur composé de *n* processeurs qui peuvent exécuter une suite d'instructions de manière asynchrone. Les déclarations d'une zone SPMD concernent un seul processeur. Ces déclarations sont dupliquées sur tous les processeurs. Les seuls identificateurs accessibles d'une zone SPMD sont les identificateurs prédéfinis par le langage et les identificateurs définis précédemment dans cette zone ou d'autres zones SPMD. Pour permettre les échanges de données entre le mode normal de programmation et le mode SPMD, les variables déclarées dans une zone SPMD sont accessibles en dehors de cette zone par une notation de sélection. Pour celà l'ensemble des entités d'un processeur sont rassemblées dans une sorte de structure, et l'ensemble des processeurs est vu comme un tableau de structures. L'identificateur en_spmd identifie l'ensemble des processeurs. Comme cet ensemble a une structure de tableau, l'ensemble des entités du processeur i est désigné par la notation en_spmd(i) et l'entité x du processeur i déclarée dans une zone SPMD est identifiée par la notation en_spmd(i)`x. Comme pour les tableaux de structures, l'ensemble des entités x déclarées SPMD est désignées par la notation en_spmd`x. Considérons les déclarations {59}, en dehors du mode SPMD, les deux variables x et T ne sont pas directement accessibles mais peuvent être identifiées par les notations suivantes :

- en_spmd(i)`x identifie la variable x du processeur i : en_spmd(i)`x.type_ ≡ entier

- en_spmd(i)`T identifie la variable T du processeur i :
  en_spmd(i)`T.type_ ≡ **tableau** [1,100] **de** entier

```
      spmd
         variable
            x : entier ;
            T : tableau [1,100] de entier ;
      fin_spmd ;
{59}
```

o en_spmd`x identifie une variable de type tableau d'entier :
  en_spmd`x.type_ ≡ **tableau processeurs de entier** ;

o en_spmd`T identifie une variable de type tableau de tableau :
  en_spmd`T.type_ ≡ **tableau processeurs de tableau** [1,100] **de entier** ;

Le type processeurs désigne l'ensemble des processeurs. Sur OPSILA : processeurs ≡ [0,15]. Normalement, toutes les entités déclarées SPMD peuvent être identifiées de cette manière. Mais, seules les valeurs et les variables peuvent être étendues en tableaux. Les types, les constantes et les patrons ne sont pas étendus. L'identification des procédures et fonctions échoue. Pour des précisions supplémentaires, voir 3.4.

13 Les instructions

Les instructions du langage Hellena sont classiques ; ce sont

o l'instruction d'affectation,
o l'instruction conditionnelle **si** ,
o les instructions d'itération **boucle** et **pour**,
o l'instruction **bloc**,
o les instructions de rupture de séquence **aller_a**, **reboucler** et **sortir_de**,
o l'instruction parallèle **pour_tout**,
o l'instruction d'appel de procédure,
o l'instruction **spmd**
o les instructions prédéfinies PSHF, Inverse_PSHF, Compression, Expansion, BTRVS, Fusion, Scatter, Gather, lire et écrire.

14 L'instruction d'affectation

Une instruction d'affectation a l'une des formes définies en {60}. La partie gauche est un nom de variable, de résultat ou un n-uplet de variables. Cette variable peut être définie par des notations d'indiçage, de sélection et d'application de patrons. L'expression en partie droite est définie de manière classique en composant des constantes, des valeurs, des variables et des appels de fonctions par des opérateurs prédéfinis ou définis par le programmeur. Pour être valide,

```
    (1)   nom_de_variable := expression ;
    (2)   n-uplet_de_variables := n-uplet ;
```
{60}

une instruction d'affectation doit respecter la contrainte suivante :

> le résultat de l'évaluation de l'expression doit être une valeur appartenant, soit au type de la variable en partie gauche, ou sinon, au type des éléments de cette variable si c'est un tableau et que l'expression produit une valeur scalaire. Les vérifications d'appartenance au type de base de la variable sont faites à la compilation. Ces vérifications sont précisées à l'exécution pour les intervalles. La seule exeption à cette règle est l'affection d'entiers à des réels qui provoque une conversion à l'exécution.

14.1 l'affectation à des scalaires

Lorsque la variable en partie gauche désigne une variable scalaire, l'expression en partie droite doit produire un scalaire. Sur les variables déclarées en {61}, les instructions d'affectation

```
      variable
         x, y : réel ;
         i, j : entier ;
         k, l : [1,100] ;
         T    : tableau [1,10] de réel ;
         S    : (s1 : entier ; s2 : réel ; ) ;
         Ti   : T(i) ;
         Ss1  : S`s1 ;
```
{61}

en {62} sont valides ; l'affectation [5] est validée à l'exécution ; les affectations [2], [3], [6] et [9]

```
   [1]    x := y ;
   [2]    x := i ;
   [3]    x := k ;
   [4]    i := k ;
   [5]    k := i ;
   [6] T(k) := i ;
   [7]    x := T(k) ;
   [8]   Ti := y ;
   [9] S`s2 := k ;
   [10] Ss1 := i ;
```
{62}

provoquent une conversion entier/réel à l'exécution.

Les instructions d'affectation en {63} sont refusées à la compilation.

```
   [11]    k := x ;        ! affectation d'un réel à un entier
   [12]    x := T ;        ! affectation d'un tableau à un scalaire
   [13]    x := S ;        ! affectation d'un n-uplet à un scalaire
{63}
```

14.2 l'affectation à des n-uplets

Pour être valide, la valeur du n-uplet en partie droite d'une affectation de n-uplets doit appartenir au type de base du n-uplet en partie gauche. Le n-uplet en partie gauche est, soit une variable de type structure, soit une variable de type n-uplet soit une notation de n-uplets de variables. Une notation de n-uplets de variables est une liste parenthésée de noms de variables ou de résultats. Soient les déclarations {64}, les instructions d'affectation en {65} sont valides.

```
      type
         complexe : ( ir, im : réel ; ) ;
      variable
         x, y, z : complexe ;
         a : ( a1, a2 : réel ; ) ;
         b : ( b1, b2 : entier  ) ;
         i : entier ;
         p, q : réel ;
{64}
```

```
   [1]     x := y ;
   [2]     x := (p,q) ;
   [3]     z := (x`re*y`re - x`im*y`im, x`re*y`im + x`im*y`re) ;
   [4] (p,q) := x ;
   [5] (p,q) := (q,p) ;
   [6]     b := (i,100) ;
   [7]     a := x ;
   [8]     y := a ;
{65}
```

Par contre les instructions en {66} sont refusées.

```
   [9]     b := a ;
   [10]    x := b ;              ! pas de conversion
   [11]    y := (p, q, p) ;      ! pas du même type
{66}
```

14.3 l'affectation à des tableaux

Deux cas peuvent se présenter :

- l'expression en partie droite produit un scalaire

C'est l'affectation d'une même valeur scalaire à tous les éléments d'un tableau. Il faut que la variable en partie gauche soit un tableau de scalaires et que l'affectation à des variables du type des éléments de ce tableau soit valide. Sur les variables déclarées en {67}, les affectations {68} sont valides. Après la dernière affectation, tous les éléments du tableau x possèdent la

```
      variable
         x : tableau [1,100]*[1,100] de [1,20] ;
         y : tableau [1,100] de tableau [1,100] de réel ;
         a : entier ;
         b : réel ;
{67}
```

```
        x := 1 ;
        x := a ;
     y(a) := b ;
        x := x(a,5) ;
{68}
```

même valeur.

- l'expression en partie droite produit un tableau

Il faut que l'affectation de valeurs du type des éléments du tableau en partie droite à des variables du type des éléments de la variable en partie gauche soit valide. Les deux tableaux doivent être structurés suivant le même nombre de dimensions et posséder le même cardinal d'indices sur les mêmes dimensions : il doivent être conformants. L'effet d'une affectation de tableaux est l'affectation terme à terme des éléments des deux tableaux. Par exemple, sur les variables déclarées en {69}, les affectations {70} sont valides. Sémantiquement, une affectation de tableau provoque le calcul de l'expression en partie droite, puis l'affectation du résultat du calcul à la variable en partie gauche. Certains problèmes peuvent apparaître lorsque les fonctions de rangement du tableau en partie gauche ne sont pas injectives.

```
      variable
         x, y : tableau [1,100]*[1,200] de réel ;
            z : tableau [0,99]*[0,99] de réel ;
            p : réel ;
{69}
```

```
   [1]   x := y ;
   [2]   x := x*y + 1 ;
   [3]   x.colonne(i) := z.diagonale*x.colonne(j) ;
   [4]   z.ligne(j) := z.ligne(j)-z.ligne(i)*p ;
{70}
```

15 L'instruction conditionnelle si

Cette instruction a une forme classique reconnue dans de nombreux langages {71}. Les

```
      si condition_1 alors
         ...
         liste instructions 1
         ...
      sinon_si condition_2 alors
         ...
         liste instructions 2
         ...
      sinon condition_f alors
         ...
         liste instructions f
         ...
      fin_si  ;
{71}
```

conditions sont des expressions à résultat booléen scalaire ; elles sont calculées dans l'ordre jusqu'à la première vérifiée. La liste d'instructions correspondante est alors exécutée et termine l'exécution de l'instruction conditionnelle. Lorsqu'aucune condition n'est vérifiée, les instructions de la partie **sinon** sont exécutées. La partie **sinon** n'est pas obligatoire. La partie **sinon_si** peut apparaître un nombre quelconque de fois.

16 Les instructions d'itération boucle et pour

De forme également classique {72}, l'instruction **boucle** exprime la répétition infinie d'une suite d'instructions. La répétition de la liste d'instructions peut être gérée explicitement par les instructions de contrôle **aller_a**, **sortir_de** et **reboucler**. Une instruction **boucle** peut être

```
      boucle
         ...
         liste instructions
         ...
      fin_boucle ;
{72}
```

préfixée par une commande **pour** qui peut prendre les deux formes définies en {73}. Une valeur

```
! forme 1
      pour i := initial sens incrément jusqu_a limite
         boucle
            ...
            liste instructions
            ...
         fin_boucle ;

! forme 2
      pour i := initial sens incrément tant_que condition_limite
         boucle
            ...
            liste instructions
            ...
         fin_boucle  ;
{73}
```

de contrôle est associée à chaque itération d'une telle boucle. Cette valeur de contrôle est désignée par l'identificateur défini en partie gauche du symbole :=. Cet identificateur est local à la construction et n'a pas a être déclaré. La valeur initiale de contrôle est définie par l'expression `initial`. Cette valeur doit être d'un type scalaire discret. Les valeurs associées aux itérations suivent une progression arithmétique de raison définie par la clé sens et une valeur d'incrément. Le sens est spécifié par l'un des mots clés **incrément** ou **décrément**. Lorsque le sens est **incrément**, l'expression incrément est la raison de la progression. Lorsque le sens est **décrément**, la raison de la progression est – incrément. La valeur de l'incrément doit être du même type de base que la valeur initiale. Le sens peut être omis et est alors pris pour **incrément**. L'incrément peut également être omis et est alors l'unité du type de base de la valeur de contrôle (1 sur les entiers). La liste d'instructions d'un corps de boucle commandée par une spécification **pour** est répétée jusqu'à ce que l'une des conditions suivantes est vérifiée :

- une instruction **aller_a** provoque un saut à l'extérieur de la boucle ;
- une instruction **sortir_de** provoque l'abandon de la boucle ;
- le sens est **incrément** est la valeur de contrôle est supérieure (strictement) à la valeur limite de la première forme ;
- le sens est **décrément** est la valeur de contrôle est inférieure (strictement) à la valeur limite de

la première forme ;

- la condition limite d'une boucle de la deuxième forme n'est pas vérifiée au lancement d'une nouvelle itération.

La valeur initiale, la valeur de l'incrément et la valeur limite sont calculées avant la première itération. La valeur de l'incrément et la valeur limite sont invariantes pendant l'exécution de la boucle. La condition limite de la deuxième forme est calculée avant l'exécution de chaque nouvelle itération.

17 L'instruction aller_a

L'instruction `aller_a` E provoque un saut à l'instruction étiquetée par E. Si l'instruction désignée est extérieure à une instruction boucle, celle-ci est abandonnée.

18 L'instruction sortir_de

L'instruction `sortir_de` E provoque l'abandon de l'instruction étiquetée par E ou de la procédure E. Le contrôle passe à l'instruction qui suit l'instruction étiquetée par E ou à l'instruction qui suit l'appel de la procédure E. Cette instruction ne peut évidemment être exécutée qu'à l'intérieur de l'instruction étiquetée ou de la procédure désignée.

19 L'instruction reboucler

L'instruction `reboucler` E, qui ne peut être exécutée qu'à l'intérieur d'une boucle étiquetée par E, provoque l'abandon de l'itération en cours. Cette instruction est équivalente à un saut derrière la dernière instruction du corps de la boucle. Notons que l'effet de l'exécution d'une instruction `reboucler` E est différent de celui d'une instruction `aller_a` E qui provoque l'initialisation d'une nouvelle boucle et donc le calcul d'une nouvelle valeur initiale, d'un nouveau pas etc.

20 L'instruction bloc

L'instruction `bloc` permet de définir des entités locales ainsi qu'une suite d'instructions qui s'appliquent à ces variables locales et aux variables globales. Cette instruction a la forme {74}. Les variables locales d'un bloc ne sont accessibles que du bloc.

```
      bloc
         ...
         liste de déclarations
         ...
      début
         ...
         liste d'instructions
         ...
      fin ;
{74}
```

21 L'instruction d'appel de procédure

Cette instruction se réduit à un nom de procédure éventuellement suivi d'une liste de paramètres effectifs {75}. Une instruction d'appel de procédure provoque l'association des

```
      p1 ( a, b) ;
      p2 ;
{75}
```

paramètres effectifs aux paramètres formels puis l'exécution de la procédure. Le nombre de paramètres effectifs doit être le même que le nombre des paramètres formels de la déclaration de la procédure. L'association des paramètres est fait dans l'ordre de déclaration et doit respecter les contraintes suivantes :

- o paramètre formel **constante** : le paramètre effectif doit être une constante incluse dans le type du paramètre formel.

- o paramètre formel **valeur** : le paramètre effectif doit être, soit une constante, soit une valeur, soit une variable dont le type est inclus dans le type de base du paramètre formel.

- o paramètre formel **variable** : le paramètre effectif doit être une variable dont le type inclut le type du paramètre formel. De plus, la valeur courante de la variable doit appartenir au type du paramètre formel au moment de l'appel.

- o paramètre formel **résultat** : le paramètre effectif doit être une variable ou un résultat dont le type inclut le type du paramètre formel.

- o paramètre formel **type** : le paramètre effectif doit être un nom de type inclus dans le type défini par le paramètre formel.

Les seules conversions implicites admises dans les associations de paramètres sont les conversions d'entiers en réels pour les paramètres formels **constante** ou **valeur**. Lorsque le type d'un paramètre formel est formel, l'association du paramètre provoque le calcul de son type

effectif.

22 L'instruction spmd

Une instruction **spmd** a la forme {76}. Le corps d'une instruction SPMD définit l'algorithme

```
      spmd
         ...
         liste instructions
         ...
      fin_spmd ;
{76}
```

déroulé par chaque processeur. Pour celà, seules les entités déclarées dans des blocs SPMD peuvent être référencées. Toutes les instructions du langage Hellena, sauf de nouvelles instructions SPMD peuvent apparaître dans une instruction SPMD. L'exécution d'une instruction SPMD provoque l'exécution des instructions de son corps par chaque processeur. L'instruction est terminée lorsque tous les processeurs ont terminé (schémas de synchronisation de type fork/join).

23 L'instruction parallèle pour_tout

Cette instruction a la forme {77}. L'exécution d'une instruction **pour_tout** définit

```
      pour_tout i1 dans D1,
      pour_tout i2 dans D2,
      ...
      pour_tout in dans Dn :
         ...
         liste instructions
         ...
      fin_pour_tout ;
{77}
```

l'exécution en parallèle d'une famille d'instructions paramétrées par des valeurs de contrôle. Par exemple, l'exécution de l'instruction {78} correspond à l'exécution en parallèle des instructions

```
      pour_tout i dans [1,n] :
         M(i,k) := ps(i) ;
      fin_pour_tout ;
{78}
```

{79}. Dans la première version d'Hellena, seules les instructions d'affectation et les instructions

```
   M(1,k):= ps(1) ; M(2,k):= ps(2) ; ... ; M(n,k):= ps(n) ;
{79}
```

conditionnelles peuvent apparaître dans le corps d'une instruction **pour_tout**. Les appels de fonctions sont également interdits. Dans une version ultérieure, il est prévu d'accepter les instructions **pour** de la première forme, les instructions **bloc**, les instructions **pour_tout**, et les instructions d'appel de procédures et de fonctions SPMD. L'affectation à des variables scalaires est interdite dans les instructions **pour_tout** : toutes les variables en partie gauche des affectations doivent donc dépendre de toutes les valeurs de contrôle de l'instruction.

Il est possible de provoquer explicitement une dépendance séquentielle entre les données lues et les données écrites d'une instruction **pour_tout** par la spécification **séquentiellement**. Ainsi, soit l'instruction {80}, La spécification **séquentiellement** impose un ordre dans les accès

```
      pour_tout i séquentiellement dans [1,n] :
         x(l(i)):=x(l(i)) + v(i)*y(c(i)) ;
      fin_pour_tout ;
{80}
```

mémoire à une même adresse et doit permettre la programmation des récurences. A première vue, une instruction **pour_tout** à laquelle la spécification **séquentiellement** est associée ressemble fortement à une instruction d'itération **pour**. La différence essentielle réside dans le fait qu'une instruction d'itération **pour** exprime une répétition d'instructions alors qu'une instruction **pour_tout** exprime l'exécution d'une famille d'instructions.

24 Les instructions prédéfinies

Des instructions prédéfinies de Hellena permettent de réaliser les opérations d'entrée/sortie et d'accéder à certaines particularités des calculateurs parallèles comme les permutations.

Les deux instructions d'entrée/sortie de base sont `lire` et `écrire`. Ces deux instructions permettent des échanges sur le support externe standard. L'instruction `lire` s'applique à des n-uplets de variables (n quelconque) et provoque l'affectation au n-uplet d'une valeur lue sur un support externe (voir exemple {81}). Dans cet exemple, l'instruction `lire` est appliquée au triplet `(x,y,T)` et provoque la lecture d'un réel qui est affecté à x, d'un deuxième réel affecté à y puis d'une suite de 200 réels qui sont affectés à T. Les tableaux sont lus ligne par ligne.

L'instruction `écrire` s'applique à un n-uplet de valeurs (n quelconque) et produit un effet inverse de l'opération `lire`. Des possibilités supplémentaires doivent être introduites en Hellena, notemment en ce qui concerne la structuration des fichiers et leur identification dans les instructions d'entrée/sortie.

Dans la version OPSILA, les instructions `lire0` et `écrire0` permettent de réaliser des échanges

```
        variable
           x, y : réel ;
           T : tableau [1,10]*[1,20] de réel ;
        début
           lire( x, y, T ) ;
        fin ;
{81}
```

avec conversion en ascii ; les instructions `lirel` et `écrirel` provoquent des conversions en hexadécimal ; les instructions `lire2` et `écrire2` permettent les échanges en binaires avec le support externe. Ces 6 instructions indiquent par un premier paramètre entier et constant le numéro de canal sur lequel l'échange est réalisé. Par l'exemple, l'instruction `écrire2 (5, T)` provoque l'écriture en binaire du tableau T sur le canal numéro 5. Un canal peut être ouvert, c'est à dire associé à un fichier du calculateur hôte, par l'instruction `ouvrir (canal, 'fichier')`. Le canal est libéré par l'instruction `fermer (canal)`. Pour plus de précisions, se reporter au manuel de référence de OPSILA.

Les instructions de permutation et de manipulation de vecteurs ont été tirées des possibilités de la machine OPSILA. Leur intérêt réside dans le fait qu'elles permettent des accès non linéaires aux vecteurs en mémoire sans nécessiter des indirections sur les calculs des adresses. Dans les définitions qui suivent, `Vd` désigne le vecteur destination, `Vs`, `Vs1` et `Vs2` désignent les vecteurs de données source, `Vc` désigne le vecteur de contrôle (vecteur de booléens), et `Vi` désigne un vecteur d'indices (entiers). Les instructions de manipulation de vecteurs actuellement définies sont :

- `PSHF (Vd,Vs)` : affecte à `Vd` le perfect-shuffle de `Vs`. `Vd` et `Vs` ont même nombre quelconque d'éléments ;
- `Inverse_PSHF(Vd,Vs)` : affecte à `Vd` l'inverse du perfect-shuffle de `Vs` ;
- `Compression (Vd,Vs,Vc)` : affecte aux premiers éléments de `Vd` et dans l'ordre les éléments de `Vs` correspondant aux valeurs vraies de `Vc` ;
- `Expansion (Vd,Vs,Vc)` : inverse de `Compression` ;
- `BTRVS (Vd,Vs)` : affecte à `Vd` le *bit-reverse* de `Vs`. La longueur de `Vd` et `Vs` doit être une puissance de 2 ;
- `Fusion (Vd,Vs1,Vs2,Vs)` : affecte à `Vd` la résultat de la fusion de `Vs1` et `Vs2` contrôlée par `Vc`. Les éléments affectés à `Vd` correspondant à des valeurs vraies de `Vc` proviennent de `Vs1` ;
- `Scatter (Vd,Vs,Vi)` : équivalent à l'instruction `∀ i, Vd(Vi(i)) := Vs(i)` ;
- `Gather (Vd,Vs,Vi)` : équivalent à l'instruction `∀ i, Vd(i) := Vs(Vi(i))` ;

25 EXEMPLE, programmation du Householder

```
programme Householder :
   type
      M : tableau discret * discret de scalaire ;
      V : tableau discret de scalaire ;
   patron
      diag de M dans V :
        pour_tout i dans type_opérande . indices . gauche :
              ( i, i )
        fin_pour_tout ;
   patron
      sous_tableau (valeur l, c : entier ; ) de M dans M :
        pour_tout i dans [ 1, l ],
        pour_tout j dans [ 1, c ] : ( i, j ) fin_pour_tout ;
   patron
      sous_vecteur ( valeur l : entier ; ) de V dans V :
        pour_tout i dans [ 1, l ] : ( i ) fin_pour_tout  ;
   patron
      ligne_triangle_inf ( valeur i : entier ; ) de M dans V :
        pour_tout j dans [ 1, i - 1 ] : ( i, j ) fin_pour_tout ;
   patron
      portion ( valeur d, f : entier ; ) de V dans V :
        pour_tout i dans [ d, f ] : ( i ) fin_pour_tout ;
   variable
      Ba : tableau [ 1, 100 ] * [ 1, 100 ] de réel ;
      Bd, Be, Be2 : tableau [ 1, 100 ] de réel ;
      Bv : tableau [ 1, 100 ] de réel ;
      n, l : entier ;
      tol, f, g, h : réel ;
début
   lire ( tol ) ;
   lire ( n ) ;
   bloc
      variable
         a : Ba . sous_tableau ( n, n ) ;
         d : Bd . sous_vecteur ( n ) ;
         e : Be . sous_vecteur ( n ) ;
         e2 : Be2 . sous_vecteur ( n ) ;
         v : Bv . sous_vecteur ( n ) ;
      début
         lire ( n ) ;
```

```
pour i := n décrément 1 jusqu_a 2
boucle
   l := i - 1 ;
   bloc
      variable
         ligne_inf_a : a . ligne_triangle_inf ( i ) ;
         portion_e : e . portion ( 1, l ) ;
         elem_sous_diag : a ( i, i - 1 ) ;
   début
      h := Somme ( ligne_inf_a * ligne_inf_a ) ;
      si h <= tol alors
         e ( i ) := 0 ;
         e2 ( i ) := 0 ;
      sinon
         e2 ( i ) := h ;
         f := elem_sous_diag ;
         si f >= 0 alors
            g := - racine_carree ( h ) ;
         sinon
            g := racine_carree ( h ) ;
         fin_si  ;
         e ( i ) := g ;
         h := h - f * g ;
         elem_sous_diag := f - g ;
         portion_e := 0 ;
         pour j := 1 incrément jusqu_a l
         boucle
            pour_tout k dans [ 1, l ] :
               v ( k ) := si k <= j alors a ( j, k )
                                    sinon a ( k, j ) fin_si  ;
               e ( k ) := e ( k ) + v ( k ) * a ( i, j ) ;
            fin_pour_tout ;
         fin_boucle ;
         portion_e := portion_e / h ;
         f := Somme ( portion_e * ligne_inf_a ) ;
         h := f / ( h + h ) ;
         portion_e := portion_e - h * ligne_inf_a ;
         pour j := 1 incrément jusqu_a l
         boucle
            pour_tout k dans [ 1, j ] :
               a ( j, k ) := a ( j, k ) - a ( i, j ) * e ( k )
                                        - a ( i, k ) * e ( j ) ;
            fin_pour_tout ;
         fin_boucle ;
      fin_si ;
```

```
        fin ;
    fin_boucle ;
    e ( 1 ) := 0 ;
    e2 ( 1 ) := 0 ;
    d := a . diag ;
    écrire ( d ) ;
    écrire ( e ) ;
    écrire ( e2 ) ;
fin ; fin.
```

COMMENTAIRES

Ce programme effectue une tridiagonalisation de matrice réelle symétrique par la méthode de Householder. Ce programme est issu d'une programmation séquentielle, et l'algorithme présenté n'est pas nécessairement le meilleur algorithme vectoriel. Le principal intérêt de ce programme est qu'il utilise la plupart des concepts de Hellena.

La taille des matrices traitées par ce programme n'est pas connue à la compilation. L'allocation dynamique de variables n'étant pas permise dans le langage Hellena, ce programme déclare des tableaux et des vecteurs d'une certaine taille maximum (100).

```
variable
    Ba : tableau [ 1, 100 ] * [ 1, 100 ] de réel ;
    Bd, Be, Be2 : tableau [ 1, 100 ] de réel ;
    Bv : tableau [ 1, 100 ] de réel ;
    n, l : entier ;
    tol, f, g, h : réel ;
```

Les dimensions du problème sont lues par l'instruction

```
lire ( n ) ;
```

la séquence d'identifications

```
variable
    a : Ba . sous_tableau ( n, n ) ;
    d : Bd . sous_vecteur ( n ) ;
    e : Be . sous_vecteur ( n ) ;
    e2 : Be2 . sous_vecteur ( n ) ;
    v : Bv . sous_vecteur ( n ) ;
```

permet de définir des matrices et des vecteurs de dimensions conformes.
Par exemple, l'identification de la figure 7 définit la sous-matrice a du tableau Ba.

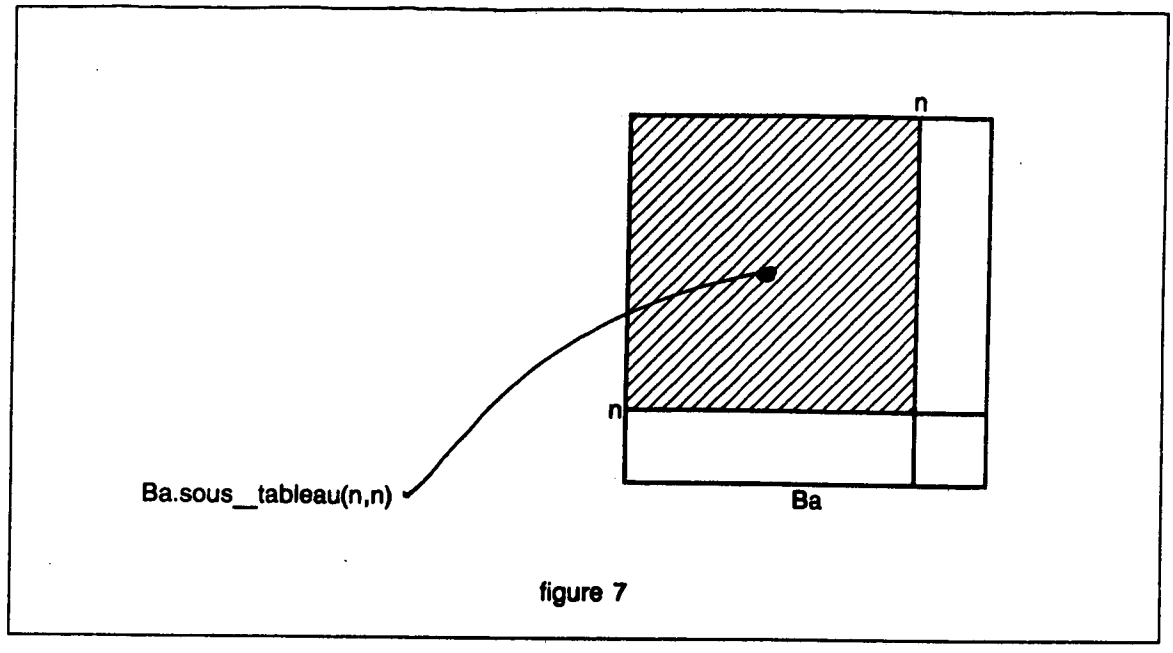


figure 7

Dans la boucle principale de ce programme, la longueur des vecteurs à accéder dépend de l'indice de la boucle. L'identification

```
ligne_inf_a : a . ligne_triangle_inf ( i ) ;
```

permet l'accès au vecteur identifié de la figure 8.

L'identification

```
elem_sous_diag : a ( i, i - 1 ) ;
```

permet l'accès à l'élément a (i, i-1).

Tout en rendant les programmes plus lisibles, ces identifications permettent une production de code machine plus efficace. En effet l'identification d'une variable entraîne le calcul d'un descripteur de rangement de l'objet et élimine la production ultérieure de séquences de code de calcul d'adresses.

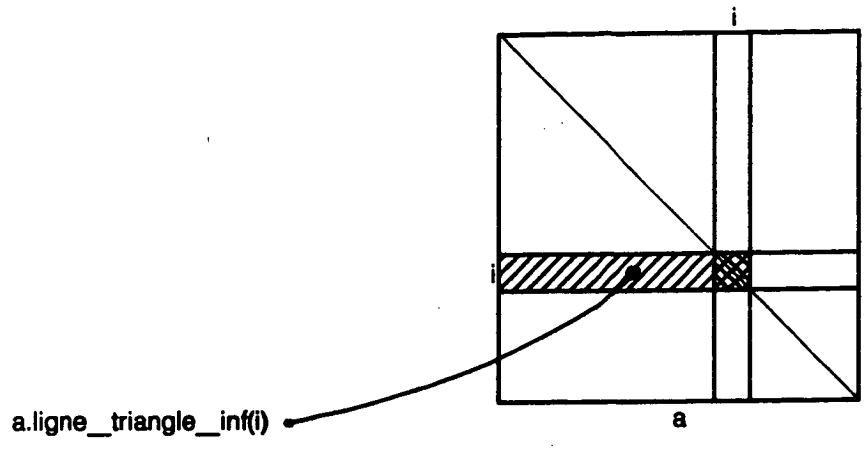


figure 8

26 Syntaxe

```
PROGRAMME ::=
   programme IDENTIFICATEUR :
      CORPS.

BLOC ::=
   bloc CORPS

BLOC_SPMD ::=
   bloc CORPS_SPMD

CORPS ::=
   PARTIE_DECLARATIVE
   INSTRUCTION_COMPOSEE

CORPS_SPMD ::=
   PARTIE_DECLARATIVE_SPMD
   INSTRUCTION_COMPOSEE_SPMD
```

```
INSTRUCTION_COMPOSEE ::=
   début
      LISTE_INSTRUCTION
   fin

INSTRUCTION_COMPOSEE_SPMD ::=
   début
      LISTE_INSTRUCTION_SPMD
   fin

PARTIE_DECLARATIVE ::=
  { CLASSE_CONSTANTES
  | CLASSE_TYPES
  | CLASSE_VARIABLES
  | CLASSE_VALEURS
  | CLASSE_SPMD
  | CLASSE_ETIQUETTES
  | CLASSE_PROCEDURES
  | CLASSE_FONCTIONS
  | CLASSE_OPERATEURS
  | CLASSE_PATRONS}

PARTIE_DECLARATIVE_SPMD ::=
  { CLASSE_CONSTANTES
  | CLASSE_TYPES
  | CLASSE_VARIABLES_PE
  | CLASSE_VALEURS
  | CLASSE_ETIQUETTES
  | CLASSE_PROCEDURES_SPMD
  | CLASSE_FONCTIONS_SPMD
  | CLASSE_OPERATEURS
  | CLASSE_PATRONS }

CLASSE_CONSTANTES ::=
   constante
     { DECLARATION_CONSTANTE }

CLASSE_TYPES ::=
   type
      { DECLARATION_TYPE }

CLASSE_VARIABLES ::=
   variable [ RANGEMENT ]
      { DECLARATION_VARIABLE }
```

```
CLASSE_VARIABLES_PE ::=
   variable
      { DECLARATION_VARIABLE }

RANGEMENT ::= UG | UCV | SPMD

CLASSE_VALEURS ::=
   valeur
      { DECLARATION_VALEUR }

CLASSE_SPMD ::=
   spmd
      PARTIE_DECLARATIVE_SPMD
   fin_SPMD

CLASSE_ETIQUETTES ::=
   étiquette
      { DECLARATION_ETIQUETTES }

CLASSE_PROCEDURES ::=
   procédure
      { DECLARATION_PROCEDURE }

CLASSE_PROCEDURES_SPMD ::=
   procédure
      { DECLARATION_PROCEDURE_SPMD }

CLASSE_FONCTIONS ::=
   fonction
      { DECLARATION_FONCTION }

CLASSE_FONCTIONS_SPMD ::=
   fonction
      { DECLARATION_FONCTION_SPMD }

CLASSE_OPERATEURS ::=
   opérateur
      { DECLARATION_OPERATEUR }

CLASSE_PATRONS ::=
   patron
      { DECLARATION_PATRON }
```

```
LISTE_IDENTIFICATEURS ::=
   IDENTIFICATEUR {, IDENTIFICATEUR }

DECLARATION_CONSTANTE ::=
   [ LISTE_IDENTIFICATEURS : EXPRESSION ] ;

DECLARATION_VALEUR ::=
   [ LISTE_IDENTIFICATEURS : EXPRESSION ] ;

DECLARATION_TYPE ::=
   [ LISTE_IDENTIFICATEURS PARAMETRES_FORMELS : DEFINITION_TYPE ] ;

DECLARATION_VARIABLE ::=
   [ LISTE_IDENTIFICATEURS : DEFINITION_TYPE [ := EXPRESSION ] ] ;
  |[ LISTE_IDENTIFICATEURS : NOM_de_variable ] ;
  |[ LISTE_IDENTIFICATEURS : EXPRESSION_VECTORIELLE ] ;

DECLARATION_ETIQUETTE ::=
   [ LISTE_IDENTIFICATEURS ] ;

DECLARATION_PROCEDURE ::=
   [ LISTE_IDENTIFICATEURS PARAMETRES_FORMELS [ récursive ] : CORPS ] ;

DECLARATION_PROCEDURE_SPMD ::=
   [ LISTE_IDENTIFICATEURS PARAMETRES_FORMELS [ récursive ] : CORPS_SPMD ] ;

DECLARATION_FONCTION ::=
   [ LISTE_IDENTIFICATEURS PARAMETRES_FORMELS
                           résultat DEFINITION_TYPE [ récursive ] :
     CORPS ] ;

DECLARATION_FONCTION_SPMD ::=
   [ LISTE_IDENTIFICATEURS PARAMETRES_FORMELS
                           résultat DEFINITION_TYPE [ récursive ] :
     CORPS_SPMD ] ;

DECLARATION_PATRON ::=
   [ LISTE_IDENTIFICATEURS PARAMETRES_FORMELS de DEFINITION_TYPE
                                               dans DEFINITION_TYPE :
     LISTE_INDICES
    |INDICE
    |POUR_TOUT_INDICE
    |REFERENCE_UNIFORME ] ;
```

```
DECLARATION_OPERATEUR ::=
   [ LISTE_IDENTIFICATEURS PARAMETRES_FORMELS dans DEFINITION_TYPE :
      EXPRESSION ] ;

PARAMETRES_FORMELS ::=
  [ ( CLASSE_PARAMETRE UN_PARAMETRE_FORMEL
      {[CLASSE_PARAMETRE] UN_PARAMETRE_FORMEL} ) ]

CLASSE_PARAMETRE ::=
  valeur | constante | variable | résultat | type

UN_PARAMETRE_FORMEL ::=
   LISTE_IDENTIFICATEURS : DEFINITION_TYPE ;

DEFINITION_TYPE ::=
   ENUMERATION
  |TABLEAU
  |ENSEMBLE
  |N_UPLET
  |ENREGISTREMENT
  |NOM_de_type

ENUMERATION ::=
   énumération LISTE_IDENTIFICATEURS

TABLEAU ::=
   tableau DEFINITION_TYPE_discret de DEFINITION_TYPE

N_UPLET ::=
    DEFINITION_TYPE * N_UPLET
   |DEFINITION_TYPE * DEFINITION_TYPE

ENREGISTREMENT ::=
   ( { DECLARATION_TYPE } )

INDICE ::=
   ( EXPRESSION {, EXPRESSION } )

LISTE_INDICES ::=
   { INDICE {, INDICE } }
```

```
POUR_TOUT_INDICE ::=
   pour_tout IDENTIFICATEUR dans DEFINITION_TYPE_discret
 {,pour_tout IDENTIFICATEUR dans DEFINITION_TYPE_discret } :
       POUR_TOUT_INDICE
   fin_pour_tout
 | INDICE

LISTE_INSTRUCTION ::=
  { INSTRUCTION }

LISTE_INSTRUCTION_SPMD ::=
  { INSTRUCTION_SPMD }

INSTRUCTION ::=
  [? ETIQUETTE ]
     |INSTRUCTION_SI
     |INSTRUCTION_CHOISIR
     |INSTRUCTION_SORTIR_DE
     |INSTRUCTION_REBOUCLER
     |INSTRUCTION_ALLER_A
     |BLOC
     |INSTRUCTION_BOUCLE
     |INSTRUCTION_POUR
     |INSTRUCTION_COMPOSEE
     |INSTRUCTION_VECTORIELLE
     |GROUPE_INSTRUCTIONS_SPMD
     |INSTRUCTION_AFFECTATION_OU_REFERENCE ] ;

INSTRUCTION_SPMD ::=
  [? ETIQUETTE ]
     |INSTRUCTION_SI_SPMD
     |INSTRUCTION_CHOISIR_SPMD
     |INSTRUCTION_SORTIR_DE
     |INSTRUCTION_REBOUCLER
     |INSTRUCTION_ALLER_A
     |BLOC_SPMD
     |INSTRUCTION_BOUCLE_SPMD
     |INSTRUCTION_POUR_SPMD
     |INSTRUCTION_COMPOSEE_SPMD
     |INSTRUCTION_VECTORIELLE
     |INSTRUCTION_AFFECTATION_OU_REFERENCE ] ;
```

```
GROUPE_INSTRUCTIONS_SPMD ::=
   SPMD
      LISTE_INSTRUCTIONS_SPMD
   fin_SPMD

INSTRUCTION_VECTORIELLE ::=
   pour_tout IDENTIFICATEUR [séquentiellement] dans DEFINITION_TYPE_discret
 {,pour_tout IDENTIFICATEUR [séquentiellement] dans DEFINITION_TYPE_discret}:
      LISTE_COMMANDE_VECTORIELLE
   fin_pour_tout  ;

LISTE_COMMANDE_VECTORIELLE ::=
  {CONDITIONNELLE_VECTORIELLE | AFFECTATION }

CONDITIONNELLE_VECTORIELLE ::=
  si EXPRESSSION alors
    { AFFECTATION }
  {sinon_si EXPRESSION alors
    { AFFECTATION }}
  [sinon
    { AFFECTATION }]
  fin_si  ;

AFFECTATION ::=
  NOM_de_variable := EXPRESSION ;

INSTRUCTION_AFFECTATION_OU_REFERENCE ::=
  NOM [:= EXPRESSION] ;

INSTRUCTION_CHOISIR ::=
   choisir EXPRESSION dans
  {cas EXPRESSION [ .. EXPRESSION ] {, EXPRESSION [ .. EXPRESSION ]} :
       LISTE_INSTRUCTION }
  [autrecas : LISTE_INSTRUCTION ]
   fin_choisir  ;

INSTRUCTION_CHOISIR_SPMD ::=
   choisir EXPRESSION dans
  {cas EXPRESSION [ .. EXPRESSION ] {, EXPRESSION [ .. EXPRESSION ]} :
       LISTE_INSTRUCTION_SPMD }
  [autrecas : LISTE_INSTRUCTION_SPMD ]
   fin_choisir  ;
```

```
INSTRUCTION_POUR ::=
   pour IDENTIFICATEUR := EXPRESSION [PAS] LIMITE
   INSTRUCTION_BOUCLE

INSTRUCTION_POUR_SPMD ::=
   pour IDENTIFICATEUR := EXPRESSION [PAS] LIMITE
   INSTRUCTION_BOUCLE_SPMD

PAS ::=
   incrément | décrément
    [ EXPRESSION ]

LIMITE ::=
   jusqu_a EXPRESSION
  |tant_que EXPRESSION_booléenne

INSTRUCTION_BOUCLE ::=
   boucle
      LISTE_INSTRUCTION
   fin_boucle

INSTRUCTION_BOUCLE_SPMD ::=
   boucle
      LISTE_INSTRUCTION_SPMD
   fin_boucle

INSTRUCTION_SI ::=
   si EXPRESSION_booléenne alors
      LISTE_INSTRUCTION
  {sinon_si EXPRESSION_booléenne alors
      LISTE_INSTRUCTION}
  [sinon
      LISTE_INSTRUCTION]
   fin_si

INSTRUCTION_SI_SPMD ::=
   si EXPRESSION_booléenne alors
      LISTE_INSTRUCTION_SPMD
  {sinon_si EXPRESSION_booléenne alors
      LISTE_INSTRUCTION_SPMD }
  [sinon
      LISTE_INSTRUCTION_SPMD ]
   fin_si
```

```
INSTRUCTION_REBOUCLER ::=
   reboucler ETIQUETTE

INSTRUCTION_SORTIR_DE ::=
   sortir_de ETIQUETTE

INSTRUCTION_ALLER_A ::=
   aller_a ETIQUETTE

EXPRESSION_ENSEMBLE ::=
  {EXPRESSION2_ENSEMBLE union} EXPRESSION2_ENSEMBLE

EXPRESSION2_ENSEMBLE ::=
  {TERME_ENSEMBLE intersection} TERME_ENSEMBLE

TERME_ENSEMBLE ::=
  {FACTEUR_ENSEMBLE *} FACTEUR_ENSEMBLE

FACTEUR_ENSEMBLE ::=
   ( EXPRESSION_ENSEMBLE )
  |NOM
  |{ EXPRESSION [..EXPRESSION] {, EXPRESSION [.. EXPRESSION]} }
  |[ EXPRESSION, EXPRESSION ]

EXPRESSION ::=
  {EXPRESSION_GENERALE "IDENTIFICATEUR"} EXPRESSION_GENERALE

EXPRESSION_GENERALE ::=
  {EXPRESSION_NUMERIQUE OPERATEUR_EXPRESSION} EXPRESSION_NUMERIQUE

EXPRESSION_NUMERIQUE ::=
  {TERME OPERATEUR_NUMERIQUE} TERME

TERME ::=
 {[OPERATEUR_UNAIRE] FACTEUR OPERATEUR_TERME} FACTEUR

FACTEUR ::=
   ( EXPRESSION )
  |EXPRESSION_CONDITIONNELLE
  |EXPRESSION_VECTORIELLE
  |NOMBRE
  |NOM
```

```
OPERATEUR_EXPRESSION ::=
   = | <> | < | > | <= | >=

OPERATEUR_NUMERIQUE ::=
   + | - | ou | ou_exclusif

OPERATEUR_TERME ::=
   * | / | divise | reste| modulo | et

OPERATEUR_UNAIRE
   - | non

EXPRESSION_CONDITIONNELLE ::=
  si EXPRESSION_booléenne
  alors EXPRESSION
  {sinon_si EXPRESSION_booléenne
   alors EXPRESSION }
  sinon EXPRESSION
  fin_si

EXPRESSION_VECTORIELLE ::=
   pour_tout IDENTIFICATEUR [séquentiellement] dans EXPRESSION_ENSEMBLE
 {,pour_tout IDENTIFICATEUR [séquentiellement] dans EXPRESSION_ENSEMBLE} :
     EXPRESSION
   fin_pour_tout

NOM ::=
  IDENTIFICATEUR {LISTE_PARAMETRES} [. NOM ]
  IDENTIFICATEUR {LISTE_PARAMETRES} [` NOM ]

LISTE_PARAMETRES ::=
   [ ( EXPRESSION {, EXPRESSION} ) ]

NOMBRE ::=
   NUMERIQUE{NUMERIQUE}

IDENTIFICATEUR ::=
   ALPHABETIQUE{ALPHABETIQUE|NUMERIQUE|TRAIT_BAS}

TRAIT_BAS ::= _
```

ALPHABETIQUE ::=

A	B	C	D	E	F
G	H	I	J	K	L
M	N	O	P	Q	R
S	T	U	V	W	X
Y	Z	a	b	c	d
e	f	g	h	i	j
k	l	m	n	o	p
q	r	s	t	u	v
w	x	y	z		

NUMERIQUE ::=

0	1	2	3	4
5	6	7	8	9

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique